



User's Guide

Version 3.2

UNIX

**ParaSoft Corporation
2031 S. Myrtle Ave.
Monrovia, CA 91016
Phone: (888) 305-0041
Fax: (626) 305-9048
E-mail: info@parasoft.com**

IMPORTANT NOTICE

LIMITED USE OF SOFTWARE LICENSE. This agreement contains the ParaSoft Corporation (PARASOFT) Limited Use Software License Agreement (AGREEMENT) which will govern the use of the ParaSoft products contained within it.

YOU AGREE TO THE TERMS OF THIS AGREEMENT BY THE ACT OF OPENING THE ENVELOPE CONTAINING THE SOFTWARE OR INSTALLING IT ON YOUR COMPUTER SYSTEM. DO NOT OPEN THE ENVELOPE OR ATTEMPT TO INSTALL THE SOFTWARE WITHOUT FIRST READING, UNDERSTANDING AND AGREEING TO THE TERMS AND CONDITIONS OF THIS AGREEMENT. YOU MAY RETURN THIS PRODUCT TO PARASOFT FOR A FULL REFUND BEFORE OPENING THE ENVELOPE OR INSTALLING THE SOFTWARE.

GRANT OF LICENSE. PARASOFT hereby grants you, and you accept, a limited license to use the enclosed electronic media, user manuals, and any related materials (collectively called the SOFTWARE in this AGREEMENT). You may install the SOFTWARE in only one location on a single disk or in one location on the temporary or permanent replacement of this disk. If you wish to install the SOFTWARE in multiple locations, you must either license an additional copy of the SOFTWARE from PARASOFT or request a multi-user license from PARASOFT. You may not transfer or sub-license, either temporarily or permanently, your right to use the SOFTWARE under this AGREEMENT without the prior written consent of PARASOFT.

DERIVED PRODUCTS. Products developed from the use of the SOFTWARE remain your property. No royalty fees or runtime licenses are required on said products.

TERM. This AGREEMENT is effective from the day you open the sealed package containing the electronic media and continues until you return the original SOFTWARE to PARASOFT, in which case you must also certify in writing that you have destroyed any archival copies you may have recorded on any memory system or magnetic, electronic, or optical media and likewise any copies of the written materials.

PARASOFT'S RIGHTS. You acknowledge that the SOFTWARE is the sole and exclusive property of PARASOFT. By accepting this agreement you do not become the owner of the SOFTWARE, but you do have the right to use the SOFTWARE in accordance with this AGREEMENT. You agree to use your best efforts and all reasonable steps to protect the SOFTWARE from use, reproduction, or distribution, except as authorized by this AGREEMENT. You agree not to disassemble, de-compile or otherwise reverse engineer the SOFTWARE.

YOUR ORIGINAL ELECTRONIC MEDIA/ARCHIVAL COPIES. The electronic media enclosed contain an original PARASOFT label. Use the original electronic media to make "back-up" or "archival" copies for the purpose of running the SOFTWARE program. You should not use the original electronic media in your terminal except to create the archival copy. After recording the archival copies, place the original electronic media in a safe place. Other than these archival copies, you agree that no other copies of the SOFTWARE will be made.

CUSTOMER REGISTRATION. PARASOFT may from time to time revise or update the SOFTWARE. These revisions will be made generally available at PARASOFT's discretion. Revisions or notification of revisions can only be provided to you if you have signed and returned the enclosed Registration Card to PARASOFT and if

your electronic media are the originals. PARASOFT's customer services are available only to registered users.

LIMITED WARRANTY. PARASOFT warrants for a period of thirty (30) days from the date of purchase, that under normal use, the material of the electronic media will not prove defective. If, during the thirty (30) day period, the software media shall prove defective, you may return them to PARASOFT for a replacement without charge. PARASOFT further allows thirty (30) days free consultation regarding the SOFTWARE, its installation and operation. After this initial period PARASOFT reserves the right to refuse further assistance unless a maintenance contract has been purchased.

SUITABILITY. PARASOFT has worked hard to make this a quality product, however PARASOFT makes no warranties as to the suitability, accuracy, or operational characteristics of this SOFTWARE. The SOFTWARE is sold on an "as-is" basis.

EXCLUSIONS. PARASOFT shall have no obligation to support SOFTWARE that is not the then current release.

LIMITATION OF LIABILITY. You agree that PARASOFT's liability for any damages to you or to any other party shall not exceed the license fee paid for the SOFTWARE.

PARASOFT WILL NOT BE RESPONSIBLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OF THE SOFTWARE ARISING OUT OF ANY BREACH OF THE WARRANTY, EVEN IF PARASOFT HAS BEEN ADVISED OF SUCH DAMAGES. THIS PRODUCT IS SOLD "AS-IS".

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

TERMINATION OF AGREEMENT. If any of the terms and conditions of this AGREEMENT are broken, PARASOFT has the right to terminate the AGREEMENT and demand that you return the SOFTWARE to PARASOFT. At that time you must also certify, in writing, that you have not retained any copies of the SOFTWARE.

ENTIRE AGREEMENT. This AGREEMENT constitutes the entire agreement between you and PARASOFT.

All brand and product names are trademarks or registered trademarks of their respective holders.

Copyright © 1998-2000

ParaSoft Corporation

2031 South Myrtle Avenue, Monrovia, CA 91016

Printed in the U.S.A, April 25, 2000

CodeWizard User's Guide

Introduction

Introduction	1
Installation	4
Contacting ParaSoft	12

Using CodeWizard

Using CodeWizard: a Simple Example	15
Using CodeWizard	20
Preventing Errors in Embedded Development	23

Customizing CodeWizard

Creating Custom Coding Standards	29
Customizing CodeWizard Options With Configuration Files	30
Customizing Output With Suppressions.....	43

Viewing Results in Insra

Insra	53
-------------	----

Items Enforced

Items Enforced	65
Effective C++ Items	67
More Effective C++ Items.....	125
Meyers-Klaus Items	141
Universal Coding Standard Items	146
User Items	207

Index	341
-------------	-----

Introduction

Welcome to CodeWizard, a unique C/C++ tool that prevents errors by automatically enforcing built-in and custom C/C++ coding standards.

Coding standards are language-specific rules that, if followed, will significantly reduce the opportunity for you to make errors. When you implement and enforce coding standards for every project every day, you can prevent errors from entering code and greatly reduce the amount of debugging that you must perform to find and fix errors after they have been introduced.

CodeWizard automatically enforces C and C++ coding standards to help you prevent errors and make your code more portable and maintainable. CodeWizard is an automatic source code analysis tool based on programming ideas from C and C++ experts. CodeWizard enforces over 170 built-in C/C++ coding standards, and any number of custom C/C++ coding standards that you create with RuleWizard, a CodeWizard feature which lets you graphically create your own coding standards or customize existing CodeWizard coding standards.

If you develop software for embedded systems, CodeWizard can help you prevent errors before your code leaves the host system. CodeWizard enforces a set of coding standards designed specifically to prevent errors in embedded software development; it also lets you easily create and enforce your own embedded development coding standards.

Supported Platforms and Compilers

The platforms and compilers supported by CodeWizard at the time this manual was printed are summarized in the table below:

Platform	OS Version	CC	cxx	g++	xLC
TrueUnix64	4.0E; 5.0		X	X	
HP10	10.20	X		X	
HP11	11.0	X		X	

Linux	RedHat 6.x (glibc 2.1)		X	
AIX4	4.3x		X	X
SGI6	6.5	X	X	
Solaris	7; 8	X	X	

The OS version listed above is the version under which CodeWizard was built. Older OS versions may work, and newer versions will generally work. If you have a different compiler, you can customize CodeWizard for your needs. Contact a ParaSoft Quality Consultant (quality@para-soft.com) for more details.

Installation

The following steps describe the process of installing CodeWizard from either:

- a physical medium such as a CD-ROM.
- an electronic archive obtained via our Web server.

Installing CodeWizard from a Web server requires no particular system privileges other than the ability to create a directory into which the software will be placed. Installing from a CD-ROM requires root privileges.

Installing CodeWizard involves the following steps:

- Step 1. Create a directory for extracting the CodeWizard distribution
- Step 2. Extract the CD-ROM contents
- Step 3. Extract the installation script
- Step 4. Install CodeWizard
- Step 5. Install a license
- Step 6. Set the PARASOFT environment variable (optional)
- Step 7. Modify your PATH
- Step 8. Modify your environment

Each of these steps is fully explained in this section.

The amount of disk space required to install CodeWizard depends on which system you are installing. The table below shows how much space is necessary for the product on each platform.

Note: The number given does not reflect how much space is necessary for the original zipped tar file or to install and/or use CodeWizard.

ARCH	Disk Space (MB)
linux	13 M
solaris	17 M

aix4	34 M
alpha4	29 M
sgi6	34 M
hp10	36 M
hp11	18 M

Step 1. Create a directory for extracting the CodeWizard distribution

Choose a location in which to install CodeWizard and make this directory with a command such as

```
mkdir /usr/local/parasoft
```

All subsequent steps must be performed in the new directory, so you should change to it now.

```
cd /usr/local/parasoft
```

Note: From now on we will assume that you have chosen to install software in a directory called `/usr/local/parasoft` as indicated above. If you actually choose a different name, you will have to modify the following commands appropriately.

If you received CodeWizard on a CD-ROM, proceed to Step 2. Extract the CD-ROM contents. If you received CodeWizard via our Web server, skip Step 2 and proceed directly to Step 3. Extract the installation script.

Step 2. Extract the CD-ROM contents

To mount the CD-ROM, you must be root.

```
su root
```

Create a /cdrom directory, if necessary

```
mkdir /cdrom
```

The actual mount command is different on each platform we support. Please use the appropriate command for your system. Note that the following commands assume that your CD-ROM drive is SCSI ID 6. If your drive is at a different SCSI location, substitute the appropriate device file name for your CD-ROM drive.

IBM AIX	<code>mount -v cdrfs -r /dev/cd0 /cdrom</code>
DEC Alpha	<code>mount -t cdfs -o rrip /dev/rz6c /cdrom</code>
HP-UX	<code>mount -F cdfs -r /dev/disk/c0t6d0 /cdrom</code>
Linux	<code>mount -t iso9660 -o ro /dev/scd0 /cdrom</code>
SCO	<code>mount -f ISO9660 -o ro /dev/cd0 /cdrom</code>
SGI IRIX	<code>mount -t iso 9660 -r /dev/scsi/sc0d610 /cdrom</code>
Solaris 2.X	<code>mount -F hsfs -r /dev/dsk/c0t6d0s2 /cdrom</code>

Change directory to the installation directory you created in Step 1.

```
cd /usr/local/parasoft
```

Then copy the tar file for your platform to the current directory with the following command. See the table at the beginning of this section for a list of valid ARCH values.

```
cp /cdrom/wizard/tar/cARCH.tar .
```

Unmount the CD-ROM with the following command.

```
umount /cdrom
```

You can now press the eject button on your CD-ROM drive to eject the CD before proceeding to Step 3.

Step 3. Extract the installation script

Extract the installation script with the command

```
uncompress -c <tar_file> | tar xvf - install
```

for a compressed tar file or

```
gzip -dc <tar_file> | tar xvf - install
```

for a gzipped tar file in which the name of the tar file supplied to you should be inserted in place of the text <archive_file>. If you are installing from a CD-ROM and your tar file is not compressed, use the following command to extract the installation script.

```
tar xvf <tar_file> install
```

You are now ready to install CodeWizard on your system using the provided installation script. Steps 4 through 9 will lead you through this procedure.

Step 4. Install CodeWizard

CodeWizard includes an installation script which will help you install CodeWizard on your system.

To run the installation script, execute the command

```
./install
```

The script first prints version and Quality Consulting information:

```
Extracting installation script...
CodeWizard Installation Script
Copyright (C) 2000 by ParaSoft Corporation
Quality Consulting is available at:
E-mail:          quality@parasoft.com
Web:            http://www.parasoft.com
```

Telephone: (626) 305-0041

Fax: (626) 305-9048

before asking you to confirm that you want to install CodeWizard into the current directory.

The CodeWizard distribution consists of the following directories and files.

- `bin.ARCH/`: CodeWizard executables
- `lib.ARCH/`: CodeWizard configuration files
- `examples`: CodeWizard example programs
- `insra`: Insra help files
- `install`: CodeWizard installation script
- `README`: CodeWizard README

Once the files have been installed, you will be led through a series of questions as the installation script configures CodeWizard for your system.

- Which compilers will be used with CodeWizard

Note: This step does not tell CodeWizard which compiler to use when instrumenting and compiling your source code. You will still need to add a `"codewizard.compiler <compiler>"` option to one of your `.psrc` files to specify which compiler should be used by CodeWizard. Therefore you should answer "yes" to these questions for any compiler which you might use with CodeWizard at some point in the future.

- Whether to send output to `stderr` or Insra (a GUI report analyzer) by default

After answering these questions, you will see the banner

```
Installation of CodeWizard 3.2 completed
```

CodeWizard is now installed on your system, but your system must be configured and a license installed before use. Steps 5, 7, and 8 are required before you can use CodeWizard.

Step 5. Install a license

After installing the necessary files, the installation script will ask if you would like to install a license to use CodeWizard. If so, the script will start `pslic`, the ParaSoft License Manager.

Note: If you get an error message from `pslic` saying that it cannot open a `.psrc` file, you should make sure that you have write permission in the `/usr/local/parasoft` (installation) directory and/or run `pslic` as superuser.

`pslic` will print out your machine and network ID numbers. You should then phone, fax, or email this information to ParaSoft. You will receive a license which you can enter using `pslic`. Once you have the license, run

```
pslic
```

Choose the **A** option to add a license.

```
(A)dd a license
```

The first item you will need to enter is the network or host ID number, which should be the same number printed by `pslic`. Next, you will be prompted to enter the expiration date, which you received from ParaSoft. Finally, enter the password you were given. To complete the license installation, select **E**.

```
(E)xit and save changes
```

Step 6. Set the PARASOFT environment variable (optional)

This step is optional.

In most cases, you will not need to set this environment variable. However, you may find it useful as a shortcut to the CodeWizard installation. Also, a tool may prompt you to set this environment variable.

To set the PARASOFT environment variable correctly, you will need to know the name of the directory in which CodeWizard has been installed on your system. Once you know this path you should define an environment variable called PARASOFT to be this pathname.

Typically, you do this by editing the file `.cshrc` in your home directory and adding a line similar to

```
setenv PARASOFT /usr/local/parasoft
```

Step 7. Modify your PATH

You must add the directory containing the executables to your execution path. Normally, you do this by adding to the definition of either the path or PATH variables, according to the shell you are using.

The directory in which the executables are located will have a name which can be derived from the type of system you are running on, and is given to you by the install script.

A typical C-shell command would be

```
set path=($path /usr/local/parasoft/bin.linux)
```

if you are planning to use one of the Linux versions of CodeWizard. Many systems have a built-in command called `arch` which can be used to determine the name of the directory to put on your path as follows

```
set path=($path /usr/local/parasoft/bin.`arch`)
```

If not installed in `/usr/local/parasoft/`, set:

```
HP      : SHLIB_PATH = lib.$ARCH
```

```
AIX     : LIB_PATH = lib.$ARCH
```

```
OTHER: LD_LIBRARY_PATH = lib.$ARCH
```

If you are in doubt as to which directory to put on your search path, ask your system administrator for help.

For sh and its derivatives (ksh, bash, zsh) you would use the following commands to modify your path:

```
PATH=/usr/local/parasoft/bin.solaris:$PATH
export PATH
```

Step 8. Modify your environment

C Shell

After modifying the appropriate configuration files, you should execute the following commands to actually modify your working environment.

```
source ~/.cshrc  
rehash
```

sh and derivatives

After modifying the appropriate configuration files, you should put your modifications in your `.profile` or `.bash_profile` (or equivalent) file, then type the following command:

```
..profile
```


Contacting ParaSoft

ParaSoft is committed to providing you with the best possible product support for CodeWizard. If you have any trouble installing or using CodeWizard, please follow the procedure below in contacting our Quality Consulting department.

- Check the manual.
- Be prepared to recreate your problem.
- Know your CodeWizard version. The number and date of your version can be seen in the banner printed when running CodeWizard on one of your files.
- If the problem is not urgent, report it by e-mail or by fax.
- If you call, please use a phone near your computer. The Quality Consultant may need you to try things while you are on the phone.

Contact Information

- **USA Headquarters**
Tel: (888) 305-0041
Fax: (626) 305-9048
Email: quality@parasoft.com
Web Site: <http://www.parasoft.com>
- **ParaSoft France**
Tel: +33 (0) 1 60 79 51 51
Fax: +33 (0) 1 60 79 51 50
Email: quality@parasoft-fr.com
- **ParaSoft Germany**
Tel: +49 (0) 78 05 95 69 60

Fax: +49 (0) 78 05 95 69 19

Email: quality@parasoft-de.com

- **ParaSoft UK**

Tel: +44 171 288 66 00

Fax: +44 171 288 66 02

Email: quality@parasoft-uk.com

Using CodeWizard: a Simple Example

You use CodeWizard by processing your source code with the special `codewizard` program in place of your normal compiler. CodeWizard will then call your compiler and proceed as usual.

While processing your code, CodeWizard will detect and report various violations of good C/C++ programming practices, such as:

- Using `new` and `delete` instead of `malloc` and `free`.
- Not calling `delete` on pointer members in destructors.
- Not making destructors virtual in base classes.
- Avoiding breaks in for loops.
- Not comparing chars to constants out of char range.
- Avoiding assignment in if statement condition.

By default, CodeWizard sends its messages to the Intra GUI. For more information on using this GUI, see “Intra” on page 53. You can also have CodeWizard send its messages to `stderr` or a report file.

The easiest way to learn how to use CodeWizard is to use it on an example program and see what it does. This section introduces one of the examples supplied with CodeWizard.

A Simple Example

To get started, make a temporary directory and copy the source code for the example to it with commands similar to

```
mkdir $HOME/codewizard
cd $HOME/codewizard
cp /usr/local/parasoftware/codewizard/examples/item11SV.C .
```

Compiling and Running Without CodeWizard

Once you have copied the example program into your current directory, you can compile it with the command

```
CC -g -o item11SV item11SV.C
```

and then run it from the shell in the normal manner.

```
item11SV
```

The program doesn't crash and doesn't print any error messages. However, this program has a serious problem in its coding structure.

Running CodeWizard

You can now run CodeWizard on this simple example to see how a violation is reported.

Although your default compiler should have been set during installation, set it now in your current directory by adding the option

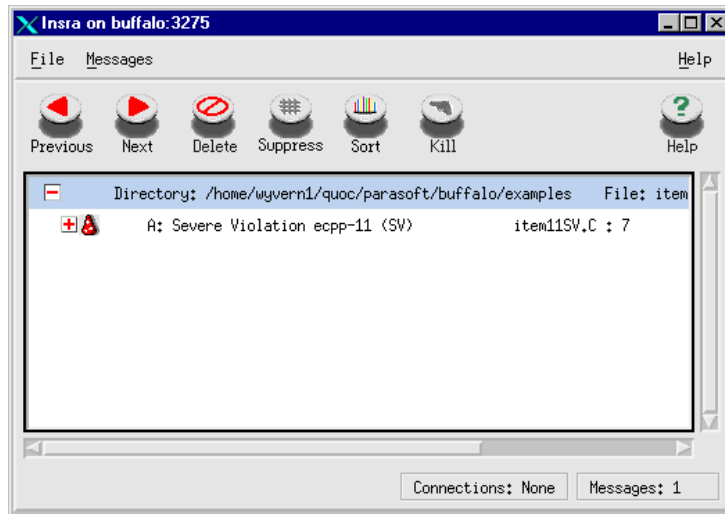
```
codewizard.compiler CC
```

to your `.psrc` configuration file. For more information about `.psrc` files and options, see “Customizing CodeWizard Options With Configuration Files” on page 30.

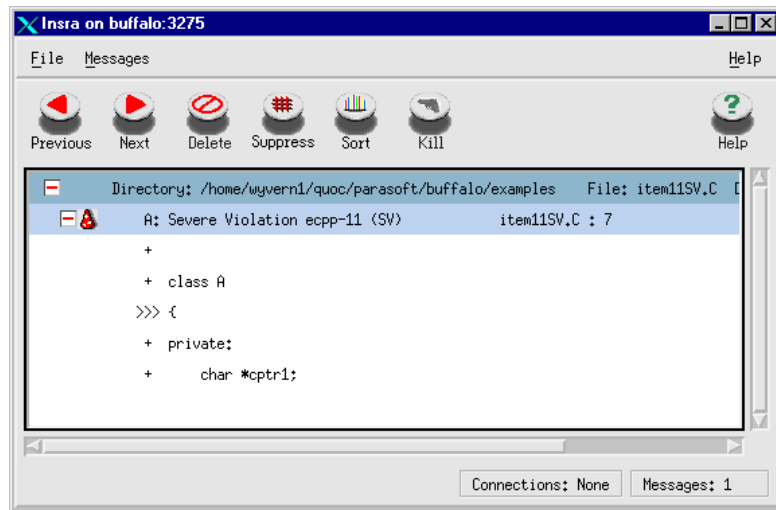
Now replace `CC` with `codewizard` on your command line:

```
codewizard -g -o item11SV item11SV.C
```

CodeWizard will then analyze the code for violations and report the following message in Instra:

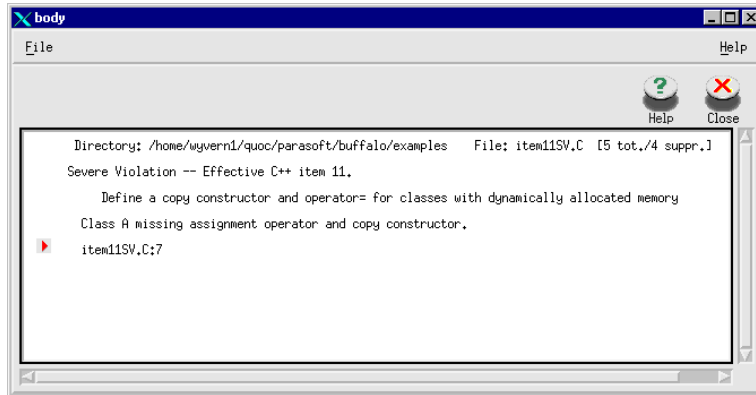


Clicking the '+' sign next to the message header will reveal the source code where the violation occurs.



Double-click the message header to open a Message window containing:

- The line of source code where the violation occurred.
- An explanation of the violation detected.



To make immediate corrections to the source code, double-click the excerpt of code shown in Insra, or click the arrow in the Message window. This will open the associated file in your selected editor, and bring you directly to the line containing the violation.

Using CodeWizard

Running CodeWizard

Analyzing a Single File

To analyze a single file, make a temporary directory and copy the source code for the example to it with commands similar to

```
mkdir $HOME/codewizard
cd $HOME/codewizard
cp /usr/local/parasoft/codewizard/exam-
ples/item11SV.C .
```

Although your default compiler should have been set during installation, set it now in your current directory by adding the option

```
codewizard.compiler CC
```

to your .psrc configuration file. For more information about .psrc files and options, see “Customizing CodeWizard Options With Configuration Files” on page 30.

Now replace CC with codewizard on your command line:

```
codewizard -g -o item11SV item11SV.C
```

Running CodeWizard From Your Makefile

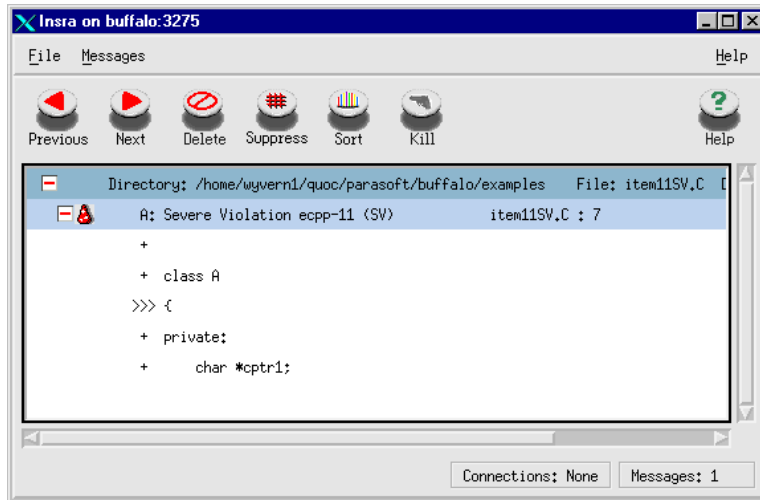
To run CodeWizard from your makefile, simply replace the name of the compiler in your makefile with `codewizard`. For example, you could use

```
make CC=codewizard
```

Results

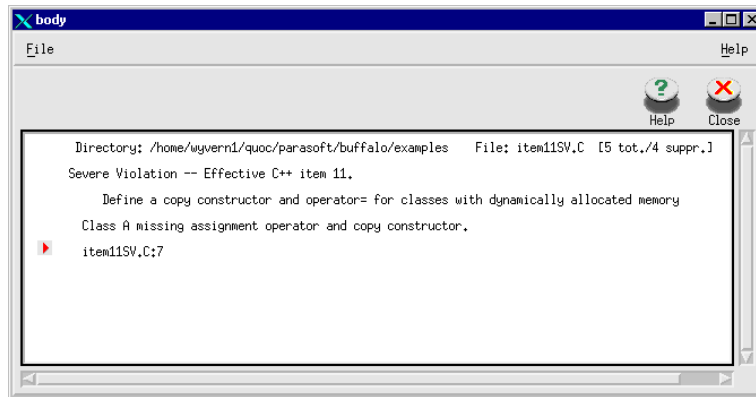
After you run CodeWizard, it will analyze the code for violations and report violation messages in Insra.

In Insra, click the “+” sign next to the message header to reveal the source code where the violation occurs.



Double-click a message header to open a Message window containing:

- The line of source code where the violation occurred.
- An explanation of the violation detected.



To make immediate corrections to the source code, simply double-click the excerpt of code shown in Insra, or click the arrow in the Message window. This will open the file in your selected editor, and bring you directly to the line containing the violation..

Note: By default, the command line will also be passed to the compiler, which will produce an executable.

If you want to use CodeWizard for checking only, you would type

```
codewizard -Zoi "codewizard.analyze_only on"
filename
```

This will analyze the code with CodeWizard, but will not pass it to the compiler. You may also do the same by including in your `.psrc` file:

```
codewizard.analyze_only on
```

Preventing Errors in Embedded Development

CodeWizard now supports embedded development by:

- Enforcing a set of coding standards designed specifically to prevent errors in C/C++ embedded software development.
- Allowing you to easily create and enforce your own embedded software development coding standards.

Enforcing Embedded Development Coding Standards

Important: To use CodeWizard on your embedded projects, you must compile your code with one of CodeWizard's supported compilers prior to running it through CodeWizard. You do not need to stop using your regular compiler; rather, you should use a supported compiler before you check your code with CodeWizard, fix the problems found, then continue to compile your code with your regular compiler. Supported compilers include:

Platform	OS Version	CC	cxx	g++	xLC
TrueUnix64	4.0E; 5.0		X	X	
HP10	10.20	X		X	
HP11	11.0	X		X	
Linux	RedHat 6.x (glibc 2.1)			X	
AIX4	4.3x			X	X
SGI6	6.5	X		X	
Solaris	7; 8	X		X	

The embedded coding standards are not enabled by default. To configure CodeWizard to enforce these coding standards:

1. Open the .psrc file (located in your CodeWizard installation directory) in any text editor.
2. Find the line that references
CodeWizard.rulesdirlist ../../cw_install_dir/rules/cwrules.txt
then replace "cwrules.txt" with "embedded.txt"

After you have performed these initial configuration steps, CodeWizard will enforce embedded development coding standards whenever you run CodeWizard.

Creating Customized Embedded Development Coding Standards

You can also extend and customize CodeWizard's set of embedded development coding standards with the RuleWizard feature that lets you graphically create your own company, personal, or target-specific coding standards as well as modify existing coding standards. For information on RuleWizard, see "Creating Custom Coding Standards" on page 29.

Embedded Development Coding Standards

The set of coding standards designed especially for embedded software developers includes:

- `ArrayElementAccess.rule`
- `AssignCharTooHigh.rule`
- `AssignCharTooLow.rule`
- `AssignUnCharTooHigh.rule`
- `AssignUnCharTooLow.rule`
- `BreakInForLoop.rule`
- `CastFuncPtrToPrimPtr.rule`
- `CastPointer.rule`
- `CastUnsigned.rule`
- `CharacterTest.rule`
- `CharCompareLeft.rule`
- `CharCompareRight.rule`
- `CommaOperator.rule`
- `ConstParam.rule`
- `ConstPointerFunctionCall.rule`
- `DeclareArrayMagnitude.rule`
- `DeclareBitfield.rule`
- `DeclareRegister.rule`
- `DeclareStaticLocal.rule`
- `DeclDimArray.rule`
- `DoWhile.rule`
- `EqualityFloatLeft.rule`
- `EqualityFloatRight.rule`

- ExplicitEnumValues.rule
- ExplicitLogicalTest.rule
- FileNameConvention.rule
- ForLoopVarAssign.rule
- FractionLoss.rule
- FuncModifyGlobalVar.rule
- FunctionSize.rule
- GlobalNameHidden.rule
- GlobalPrefixExclude.rule
- GlobalVarFound.rule
- HeaderInitialization.rule
- IfAssign.rule
- IfElse.rule
- InitCharOutOfRange.rule
- InitPointerVar.rule
- InitUnCharOutOfRange.rule
- LocalVariableNames.rule
- LongConst.rule
- ManyCases.rule
- ModifyInCondition.rule
- NameBool.rule
- NameConflict.rule
- NameConstantVar.rule
- NameDataMember.rule
- NameDouble.rule
- EnumNamingConvention.rule
- NameEnumType.rule

- NameFloat.rule
- NameGlobalVar.rule
- NameInt.rule
- NameIsFunction.rule
- NameLongInt.rule
- NamePointerVar.rule
- NameShortInt.rule
- NameSignedChar.rule
- NameString.rule
- NameStructType.rule
- NameUnsignedChar.rule
- NameUnsignedInt.rule
- NameVariable.rule
- NamingStructUnionMembers.rule
- NonScalarTypedefs.rule
- NumberFunctionParam.rule
- PassByValue.rule
- PointerParamDereference.rule
- ReferenceInitialization.rule
- SourceFileSize.rule
- SourceNamingConvention.rule
- TooManyFields.rule
- UnionFieldNotDefined.rule
- UnnecessaryEqual.rule
- UnusedLocalVar.rule
- UnusedParameter.rule
- UsePositiveLogic.rule

- `ImplicitUnsignedInit.rule`
- `BitwiseInCondition.rule`
- `ExprInSizeof.rule`
- `UnreachableCode.rule`
- `SingularSwitchStatement.rule`

Creating Custom Coding Standards

(Version 3.x Only)

Custom coding standards are rules that are specific to a certain development team, or even a certain developer. There are two types of custom coding standards: company coding standards and personal coding standards. Company coding standards are rules that are specific to your company or development team. For example, a rule that enforces a naming convention unique to your company would be a company coding standard. Personal coding standards are rules that help you prevent your most common errors. Every time you make an error, you should determine why it occurred, then design a personal coding standard that prevents it from reoccurring. If you do this religiously, you will quickly build a set of coding standards that can prevent your most common errors from entering your code.

CodeWizard's RuleWizard feature lets you design custom coding standards by graphically expressing the pattern that you want CodeWizard to look for during automatic coding standard enforcement. Rules are created by selecting a main "node," then adding additional elements until the rule expresses the pattern that you want to check for. Rule elements are added by pointing, clicking, and entering values into dialog boxes. You can also use this tool to customize many of CodeWizard's built-in coding standards.

To launch RuleWizard, type `rulewizard` at the prompt.

A complete user's guide is available in RuleWizard; to access this user's guide, choose **Help> RuleWizard Documentation** in the RuleWizard GUI. Refer to the RuleWizard User's Guide for information about creating, enforcing and enabling/disabling rules.

Customizing CodeWizard Options With Configuration Files

CodeWizard reads options from files called `.psrc`, which may exist at various locations in the file system. These options control the behavior of CodeWizard and other ParaSoft tools. The files are processed in the order specified below:

- The file `.psrc` in the appropriate lib and compiler subdirectories of the main ParaSoft installation directory, e.g.
`/usr/local/parasoft/lib.linux/CC/.psrc`
- The file `.psrc` in the main installation (`$PARASOFT`) directory, e.g.
`/usr/local/parasoft/.psrc`
- A file `.psrc` in your `$HOME` directory, if it exists.
- A file `.psrc` in the current working directory, if it exists.
- Files specified on the command line with `-Zop` and/or `-Zoi` to the `codewizard` command in the order present on the command line.

In each case, options found in later files override those seen earlier. All files mentioned above will be processed and the options set before any source files are processed.

Typically, compiler-dependent options are stored in the first location, site-dependent options are stored in the second location, user-dependent options are stored in the third location, and project-dependent options are stored in the fourth location. `-Zop` is commonly used for file-dependent options, and `-Zoi` is commonly used for temporary options.

Format

CodeWizard configuration files are simple ASCII files created and modified with a normal text editor.

Entries which begin with the character `#` are treated as comments and ignored, as are blank lines.

All keywords can be specified in either upper or lower case, and embedded underbar characters (`_`) are ignored. Arguments can normally be entered in either case, except where case has specific meaning, such as in directory or file names.

If a line is too long, or would look better on multiple lines, you can use the `\` character as a continuation line.

Option Values

The following sections describe the interpretation of the various parameters. Some options have default values, which are printed in the following section in **boldface**.

Filenames

Several CodeWizard options can specify filenames for various configuration and/or output files. You may either enter a simple filename or give a template which takes the form of a string of characters including tokens such as those in the following table, environment variables, and the `~` character. Each of these is expanded to indicate a certain property of your program as indicated in the following table.

Key	Meaning
%a	Machine architecture on which you are running, e.g., linux, rs6000, hp, etc.
%c	Abbreviated name of the compiler you are using, e.g. CC, gcc, xIC etc.
%d	Time of program compilation in format: YYYYMMDDHHMMSS
%p	Process I.D.
%r	CodeWizard version number, e.g. 3.2

%R CodeWizard version number without `.' characters, e.g., version 3.2 becomes 32.

Thus, the template

```
report_file $HOME/codewizard/%a/%c/foo.C.out
```

will cause CodeWizard to write its output to a file with a name such as

```
/usr/me/codewizard/linux/CC/foo.C.out
```

in which the environment variable HOME has been replaced by its value and the `%a' and `%c' tokens have been expanded to indicate the architecture and compiler name in use.

Using -Zop and -Zoi

On the command line, the -Zop and -Zoi options are processed from left to right, *after* all other .psrc files, and before processing any source code. Therefore, the following command line would tell CodeWizard to analyze all the files using the CC compiler.

```
codewizard -Zoi "compiler g++" foo.C
           -Zop foo.def foo2.C -Zoi "compiler g++"
           foo3.C -Zoi "compiler CC"
```

```
foo.def:
compiler CC
```

Options passed using -Zoi do not require the codewizard prefix as options passed in .psrc or -Zop files do. With -Zoi, the appropriate prefix is implicitly added using the name of the command to which the -Zoi is being passed.

Options Used by CodeWizard

codewizard.analyze_only [on|off]

Specifies whether CodeWizard should pass the file to the compiler or just check for item violations (the default). Turning this option to off tells CodeWizard to pass the file on to the compiler after checking for item violations.

codewizard.c_as_cpp [on|off]

Specifies whether files with the `.c` extension should be treated as C++ source code. With this option off, CodeWizard will treat files with the `.c` extension as C code only. If you use C++ code in `.c` files, you should turn this option on.

codewizard.compiler compiler_name

Specifies the name of an alternative compiler, such as `gcc`. If your new compiler is not recognized by CodeWizard, you may have to set the `compiler_acronym` option. This option overrides the `compiler_*` options: `compiler_acronym`, `compiler_c`, `compiler_cpp`, and `compiler_default`. The indicated compiler will be called every time `codewizard` is called.

codewizard.compiler_acronym abbreviation

Specifies the colloquial name of an alternative compiler, such as `gcc`. This name is used to locate the appropriate `.psrc` files. It does not indicate which compiler will actually be called to compile source files (see the other `compiler_*` options). This option overrides the `compiler_c`, `compiler_cpp`, and `compiler_default` options. In addition, this option must be placed *after* the active compiler option. The order must be

```
compiler c89
```

```
compiler_acronym cc
```

and not vice-versa. This option defaults to the last path component of your compiler's name, with the exception of the System V compatibility compiler on Sun workstations (`/usr/5bin/cc`), which has the acronym `5cc`.

codewizard.compiler_c C_compiler_name

Specifies the name of the default C compiler, such as `gcc`. This compiler will be called for any `.c` files. The default is `cc`. This option is overridden by the `compiler` and `compiler_acronym` options.

codewizard.compiler_cpp C++_compiler_name

Specifies the name of the default C++ compiler, such as `g++`. This compiler will be called for any `.cc`, `.cpp`, `.cxx`, and `.C` files. The default is platform dependent: `cxx` for Alpha, `g++` for Linux, `xlc` for RS/6000, and `cc` for other platforms. This option is overridden by the `compiler` and `compiler_acronym` options.

`codewizard.compiler_default` [c|cpp]

Specifies whether the default C or C++ compiler should be called to link when there are no source files on the link line. This option is overridden by the `compiler` and `compiler_acronym` options.

`codewizard.compiler_flags` flags

CodeWizard will add the flags whenever you compile your program (but they will not be passed to the preprocessor).

`codewizard.compiler_keyword` [*|const|inline|signed|volatile] keyword

Specifies a new compiler keyword (by using the `*`) or a different name for a standard keyword. For example, if your compiler uses `__const` as a keyword, use the option

```
compiler_keyword const __const
```

`codewizard.compiler_options` keyword value

Specifies various capabilities of the compiler in use, as described in the following table.

Keyword	Value	Meaning
<code>ansi</code>	None	Assumes compiler supports ANSI C (default).
<code><type> bfunc</code>	Function name	Specifies that the indicated function is a compiler "built-in" that is dealt with specially. The optional type keyword specifies that the built-in has a return type other than <code>int</code> . Currently, only <code>long</code> , <code>double</code> , <code>char *</code> , and <code>void *</code> types are supported.

btype	Type name	Specifies that the named type is a "built-in" that is treated specially by the compiler.
bvar	Variable name	Specifies that the named variable is a "built-in" that is treated specially by the compiler.
esc_x	Integer	Specifies how the compiler treats the <code>'\x'</code> escape sequence. Possible values are: 0: treat <code>'\x'</code> as the single character <code>'x'</code> (Kernighan & Ritchie style). -1: treat as a hex constant. Consume as many hex digits as possible. >0: treat as a hex constant. Consume at most the given number of hex digits.
for_scope	nested not-nested optional	Specifies how <code>for(int i; ...; ...)</code> is scoped. Possible values are: nested: New ANSI standard, always treat as nested. notnested: Old standard, never treat as nested. optional: New standard by default, but old-style code is detected and treated properly (and silently).
knr	None	Assumes compiler uses Kernighan and Ritchie (old-style) C.
loose	None	Enables non-ANSI extensions (default).
namespaces	None	Specifies that <code>namespace</code> is a keyword (default).
no-namespaces	None	Specifies that <code>namespace</code> is not a keyword.
promote_long	None	If this option is found, integral data types are promoted to <code>long</code> in expressions, rather than <code>int</code> .

size_t	d,ld,u, lu	Specifies the data type returned by the <code>sizeof</code> operator, as follows: d = int, ld = long, u = unsigned int, lu = unsigned long.
strict	None	Disables non-ANSI extensions (compiler dependent).
xfunctype	Function name	Indicates that the named function takes an argument which is a data type rather than a variable (e.g., <code>alignof</code>).

`codewizard.error_format` string

Specifies the format for error message banners generated by CodeWizard. The string argument will be displayed as entered with the macro substitutions taking place as shown in the following table. The string may also contain standard C formatting characters, such as `'\n'`

Key	Expands to
%c	Error category (and sub-category if required).
%C	Name of the class in which the error occurred.
%d	Date on which the error occurs. (DD-MON-YYYY).
%f	Filename containing the error.
%F	Full pathname, including directories, or the file containing the error.
%h	Name of the host on which the application is running.
%l	Line number containing the error.
%p	Process ID of the process incurring the error.

%t Time at which the error occurred. (HH:MM:SS).

%v The severity level of a violation.

%V Brief description of violation.

codewizard.max_call_stack_depth [1|2|3|...]

Specifies how deeply CodeWizard should traverse the call stack in checking for violations. This option currently only affects item 16, which suggests that `operator=` should assign all data members. With the default setting of 1, CodeWizard will see all assignments made in `operator=` and any function called by `operator=`. However, if one of the functions called by `operator=` calls another function which assigns a data member, CodeWizard will not see this assignment and will report a violation. With this option set to 2, CodeWizard will see the assignment and will not report a violation.

codewizard.optionfile [option] file

Directs CodeWizard to read the specified file as a `.psrc` file. If [option] is specified, then it interprets every line in the file as arguments to the option.

codewizard.preprocessor_flag flag

Specifies a flag or flags that can safely be passed to the preprocessor. Any flags on CodeWizard command lines that are not listed in a `preprocessor_flag` statement will be stripped from the command before invoking the preprocessor. The list of pre-processor flags is usually maintained in the file

```
<install_dir>/lib.$ARCH/$COMPILER/.psrc
```

Flags specified by this option will be passed to the preprocessor only, unless they are also specified in a `preprocessor_propagate_flag` option.

codewizard.preprocessor_propagate_flag flag

Specifies a flag or flags that should be passed to both the preprocessor *and* the compiler. If the flag is not listed in a `preprocessor_flag`

option, this option will be ignored.

`codewizard.report_banner` [**on**|off]

Controls whether or not a message is displayed on your terminal, reminding you that error messages have been redirected to a file.

`codewizard.report_file` [filename|**insra**]stderr]

Specifies the name of the report file. Environment variables and various pattern generation keys may appear in filename. (See “Filenames” above.) Use of the special filename `insra` tells CodeWizard to send its output to the GUI, Insra.

`codewizard.report_overwrite` [**on**|off]

If set to `off`, error messages are appended to the report file rather than overwriting it on each run.

`codewizard.rulefile` filename (**Version 3.0 only**)

Specifies a file that lists the user-defined `.rule` files that you want CodeWizard to enforce. CodeWizard will assume that this file is in [CodeWizard install dir]/rules. You may have more than one rulefile entry.

`codewizard.rulesdirlist` filename (**Version 3.1 and higher only**)

Specifies the path to a text file that lists a set of user-defined `.rule` files that you CodeWizard to enforce; the rules listed in the text file must be contained in the same directory as the text file. By default, this option is set to `<codewizard_install_dir>/rules/cwrules.txt`. You may have more than one rulesdirlist entry.

`codewizard.ruleset` NAME PATH

NAME is the identifier. PATH is the absolute location of shared libraries. Set PATH to empty to not load libraries. Basic libraries that come with CodeWizard are *Effective C++*, *More Effective C++*, *MK*, and *UCS*.

`codewizard.suppress` [class <name>] [file <name>] [item <number>] [type I|PSV|PV|SV|V] [all]

Suppresses violations matching the indicated suppression codes.

Only one of each type of suppression category is allowed per line. For example,

```
suppress class foo file foo.c item ecpp-2 type I
```

is allowed, but

```
suppress class foo class bar
```

is not. Of course, multiple lines are permissible, with later unsuppress options superseding earlier suppress options (and vice-versa) if there are common categories. The all category suppresses all messages. It can be used in conjunction with subsequent unsuppress options to turn on a limited set of messages.

codewizard.suppress_c_structs [on|off]

Suppresses item violations in C structs using a heuristic method. Turning this option to off will disable this method, and may result in some incorrect item violations.

codewizard.suppress_file string

This option tells CodeWizard not to report any violations from files which match the string. The string should be a glob-style regular expression and should include the full path. This option overrides any suppress options. To see violations from the files covered by this option, you will need to remove it. The unsuppress option will not unsuppress violations from these files.

codewizard.suppress_warning code

Suppresses parser warnings matching the indicated code. The code should match the numerical code CodeWizard prints with the warning message you would like to suppress. The codes correspond to the chapter, section, and paragraph(s) of the draft ANSI standard on which the warning is based. For example, to suppress the warning:

```
Warning:12.3.2-5: return type may not be
```

```
specified for conversion functions
```

add the following line to your `.psrc` file.

```
suppress_warning 12.3.2-5
```

`codewizard.temp_directory` path

Specifies the directory where CodeWizard will write its temporary files, e.g. **/tmp**. The default is the current directory.

`codewizard.unsuppress` [class <name>] [file <name>] [item <number>]
[type I|PSV|PV|SV|V] [all]

Unsuppresses violations matching the indicated suppression codes. For more details on how to properly use this option, see the `suppress` option .

Options Used by Insra

Running Insra

`insra.body_background_color` [**White**|color]

Specifies the color Insra will use for the message body area background.

`insra.body_font` [**Fixed**|font]

Specifies the font Insra will use for the message body text. On some systems, e.g. SGI, the Fixed font is much too large. This option can be used to select a smaller font.

`insra.body_height` [0|1|2|...|**80**|...]

Specifies the starting height of the Insra message body area in number of rows of visible text. This may be modified while Insra is running using the standard Motif controls.

`insra.body_text_color` [**Black**|color]

Specifies the color Insra will use for the message body text.

`insra.body_width` [0|1|2|...|**80**|...]

Specifies the starting width of the Insra message body area in number of columns of visible text. This may be modified while Insra is running using the standard Motif controls. If this option is set to a different value than `header_width`, the larger value will be used.

`insra.header_background_color` [**White**|color]

Specifies the font Insra will use for the message header area background.

`insra.header_font` [**Fixed**|font]

Specifies the font Insra will use for the header body text. On some systems, e.g. SGI, the Fixed font is much too large. This option can be used to select a smaller font.

`insra.header_height` [0|1|2|...|**80**|...]

Specifies the starting height of the Insra message header area in number of rows of visible text. This may be modified while Insra is running using the standard Motif controls.

`insra.header_highlight_color` [**LightSteelBlue2**|color]

Specifies the color Insra will use to indicate the currently selected message or session header in the message header area.

`insra.header_highlight_text_color` [**Black**|color]

Specifies the color Insra will use for the text of the currently selected message or session header in the message header area.

`insra.header_session_color` [**Black**|color]

Specifies the color Insra will use to indicate a session header.

`insra.header_text_color` [**Black**|color]

Specifies the color Insra will use to for session header text.

`insra.header_width` [0|1|2|...|**80**|...]

Specifies the starting width of the Insra message header area in number of columns of visible text. This may be modified while Insra is running using the standard Motif controls.

ning using the standard Motif controls. If this option is set to a different value than `body_width`, the larger value will be used.

`insra.port [3255|port_number]`

Specifies which port Insra should use to communicate with CodeWizard.

`insra.toolbar [on|off]`

Specifies whether Insra's toolbar is displayed. All toolbar commands can also be chosen from the menu bar.

`insra.visual [xterm -e vi +%l %f|emacs +%1 %f]`

Specifies how Insra should call an editor to display the line of source code causing the error. Insra will expand the `%l` token to the line number and the `%f` token to the file name before executing the given command. It is important to include the full path of any binary that lives in a location not on your path. Setting this option with no command string disables source browsing from Insra.

Customizing Output With Suppressions

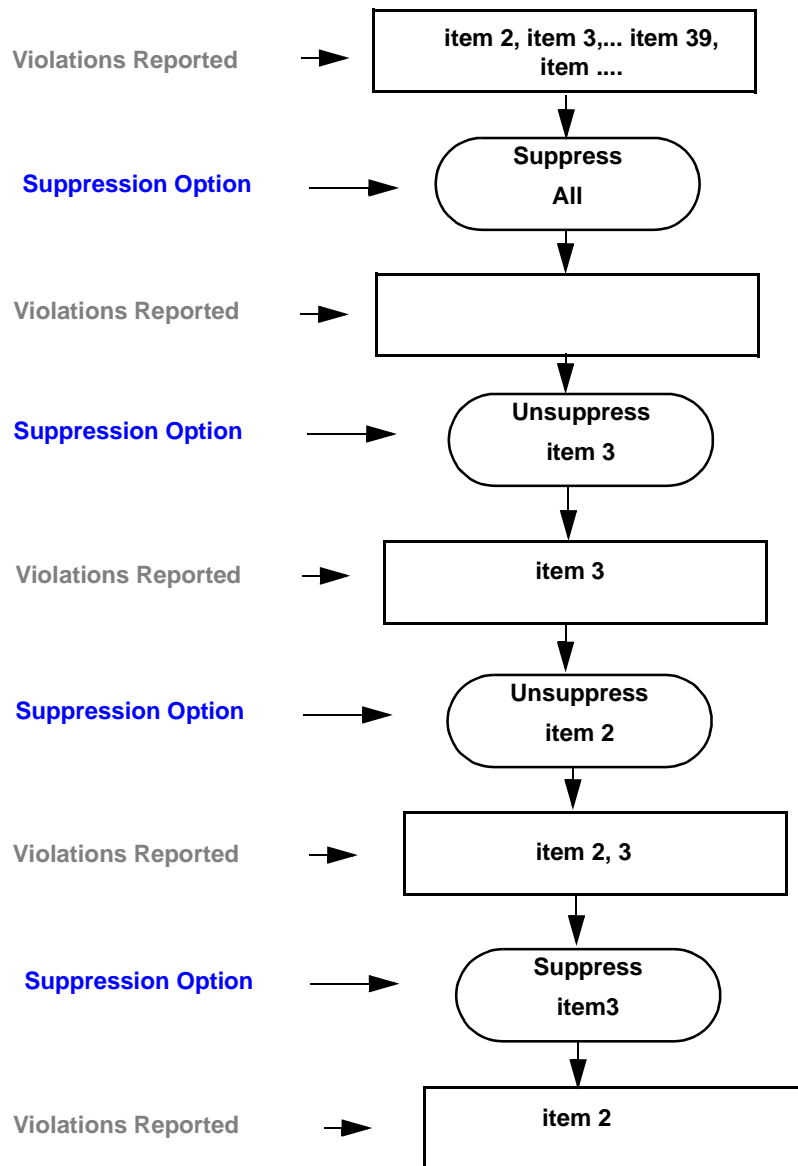
Suppressions stop CodeWizard from enforcing certain coding standards. You can use suppressions to tailor CodeWizard's output to your development team and the current project. By default, CodeWizard is configured to suppress all violations in the Universal Coding Standard category and all violations that have an Informational severity, and to unsuppress any enabled User items.

Topics discussed in this section include:

- “How Suppression Options Work”
- “Adding a Suppression Option”
- “Deleting a Suppression Option”
- “Moving a Suppression Option”
- “Determining Where Suppression Options Are Saved”
- “An Example With Suppressions”

How Suppression Options Work

Suppression options are read from top to bottom when CodeWizard determines which item violations will be reported and which will be suppressed. Because of this, you can think of a series of suppression options as a series of sieves that remove options as they pass through from top to bottom of the suppressions page. Conversely, unsuppress options can restore violations that were screened out by earlier suppress options. This can get complicated when many suppress and unsuppress options are being used concurrently, but offers a very powerful method of controlling the messages that will be displayed. The following flow chart illustrates one simple example.



Adding a Suppression Option

You can enter suppressions in the Suppressions Control Panel, which is accessible by clicking the **Suppress** button in Insra when Insra is running and contains violations.

To add a new suppression option, place your cursor in the table row where you want the suppression option inserted, then click the **Insert** button. As is, this new option will suppress all violations reported by CodeWizard.

To determine exactly what violations this option suppresses (or unsuppresses), edit its various fields. Each field can be modified to restrict the messages that will be suppressed or unsuppressed. The available fields are:

- The Action Field
- The Persistence Field
- The Item Field
- The Type Field
- The Class Field
- The File Field
- The Notes Field

The Action Field

This field specifies whether the violation message listed is to be suppressed (as indicated by a speaker with an X through it) or unsuppressed (indicated by a speaker with no X through it). Double clicking on the field toggles between suppressing and unsuppressing the violation message.



The Persistence Field

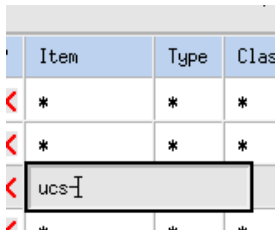
This field specifies whether the suppression option will be saved to the `.psrc` file under which it is listed. Double-clicking this field toggles it from persistent (the field is checked) to temporary (the field is unchecked). Options marked as persistent will be added to the appropriate `.psrc` files when the **Save** button is clicked on. Options marked as temporary will be discarded.

Options with an X in this field cannot be made persistent, either because they are hardwired or because the file in which it is placed is not writable. New suppression options are marked as persistent by default.

The Item Field

The **Item** field lets you determine the item violation that you would like to suppress or unsuppress. By default, there is an asterisk present, which signifies that all items are suppressed or unsuppressed (depending on which option you selected in the **Action** field).

To enter the item number, double-click in the Item field. A text field will open. Enter the item that you want to suppress or unsuppress in the text field, then press **Enter**.

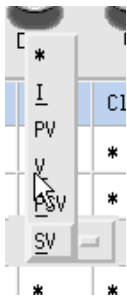


	Item	Type	Class
<	*	*	*
<	*	*	*
<	ucs		
<			

* matches all item numbers. The prefix "ecpp" stands for Effective C++, "mecpp" for More Effective C++, "mk" for Meyers-Klaus, and "ucs" for Universal Coding Standards; "user" is used to indicate rules created in RuleWizard or built-in rules that can be modified in RuleWizard.

The Type Field

The **Type** field allows you to select the type of violation for which messages will be suppressed or unsuppressed. To select the type, double-click the **Type** field. A pull-down menu will appear.



This menu contains the following options (in increasing order of seriousness):

- **I** Informational (suppressed by default)
- **PV** Possible Violation
- **V** Violation
- **PSV** Possible Severe Violation
- **SV** Severe Violation

* Matches all types

The Class Field

The **Class** field allows you to enter a class for which messages will be suppressed or unsuppressed. A field containing only "*" will match all classes. To enter the class, double-click the **Class** field. A text field will open. Enter the desired class name in the text field, then press **Enter**.

Severity	Class	File
*		*
*		de
*		/t
*		/t

The File Field

This field allows you to enter a file for which messages will be suppressed or unsuppressed. A field containing only "*" will match all files. To enter a file name, type it in the **File** field, or double-click the **File** field and select a file using the file chooser that opens.

File	No
*	
demo.C	
item11SV.C	aso

The Notes Field

This field allows you to create notes to explain the purpose of the current suppression item. To enter a note, double-click the **Note** field. A text field will open. Enter the desired note in the text field, then press **Enter**.

Deleting a Suppression Option

To delete a suppression option, click it to select it, then click the **Delete** button.

Once this deletion has been applied, the effects of this option will be gone. This change will appear in Insra immediately.

Moving a Suppression Option

You may move individual suppression options into new locations on the suppressions page by clicking the **Up** and **Down** buttons. This will affect how CodeWizard reports item violations. For more information on this topic see “How Suppression Options Work” above.

Determining Where Suppression Options Are Saved

The headers in Insra show the various locations in which `.psrc` files reside. Insra will display the suppression options as read from each file

under the appropriate header. When you add a new option using the **Insert** button, it will be inserted below the currently selected option. You can then move it into the file in which you would like it saved, or mark it as temporary by double-clicking the persistence field (see above). There are two special locations where suppression options may reside other than actual `.psrc` files: hardwired options and command line options. The former are set internally by CodeWizard, and therefore cannot be permanently changed. They can be edited and/or removed in the window temporarily, however. The latter are options passed using the `-Zop` and `-Zoi` options on the command line. These options, like hardwired options, cannot be made persistent, but can be moved into a `.psrc` file if you decide that you want to make them permanent.

An Example With Suppressions

This section will build on the simple example and show how to use suppressions.

For example, let's assume that you have just run the simple example, then have decided that you want to unsuppress all violations of rule `ucs-19`. This rule states that a class that has pointer members shall have an `operator=` and a copy constructor. Adhering to this rule prevents memory leaks and data corruption. If you do not define a copy constructor, the default constructor will be used, which is usually *not* the desired behavior when a class has pointer members.

To have CodeWizard report all violations of `ucs-19` that it finds in all files that it analyzes, perform the following steps:

1. To open the Suppressions Control Panel, click the **Suppress** button in Insra when Insra contains one or more violations. The Suppressions Control Panel will open.
2. Place your cursor in the last row of suppressions, then click the **Insert** button. An additional row of asterisks will appear.
3. In the new suppression option's **Action** field, choose **Unsuppress** by clicking the speaker icon until the speaker does not have an X over it.
4. In the new suppression's **Item** field, enter `ucs-19`.

5. Save your new suppression by clicking **Save**.

The Insra GUI will then report that a violation of the ucs-19 rule occurred in item11SV.C.

Insra

Insra is a graphical user interface for displaying violation messages generated by CodeWizard. The messages are summarized in a convenient display, which allows you to quickly navigate through the list of violation messages, suppress messages, invoke an editor for immediate corrections to the source code, and delete messages as violations are fixed.

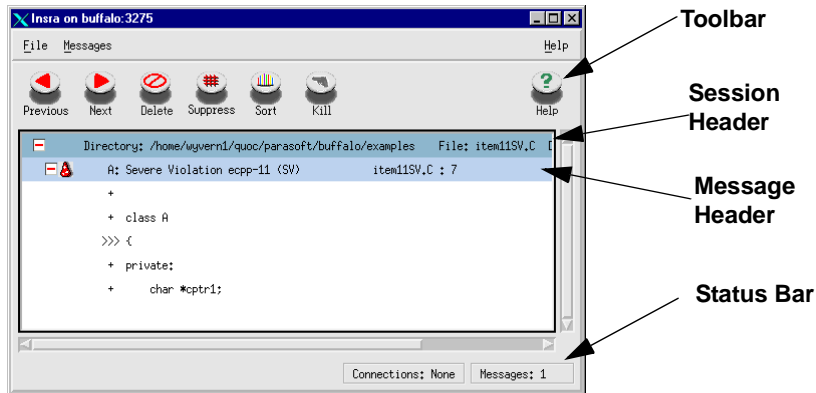
Topics discussed in this section include:

- “The Insra Display”
- “Sending Messages to Insra”
- “Viewing and Navigating”
- “Deleting Messages”
- “Suppressing Messages”
- “TheSuppressions Tool Bar”
- “Miscellaneous Issues”
- “Help”
- “Setting Preferences”
- “Troubleshooting”

The Insra Display

Status Bar

During analysis, CodeWizard makes a connection to Insra each time a violation is detected. The status bar will report the number of violation messages currently being displayed and the number of active connections. An active connection is denoted by a yellow star to the left of the session header. A connection will remain active as long as the program is still compiling/running. Insra will not allow you to delete a session header as long as its connection remains active, and you may not exit Insra until all connections have been closed.



Tool Bar

The tool bar allows you to:

- Scroll up and down through messages.
- Delete selected messages as bugs are fixed.
- Suppress violations detected by CodeWizard.
- Sort messages by order (time) reported, violation category, or directory and file.
- Kill the selected active connection.

Message Header Area

The message header area contains session headers and message headers for programs currently connected to Insra.

Session Header

When the first violation is detected for a particular compilation or execution, a session header is sent to Insra. The session header includes the following information:

- Compilation/execution
- Source file/program
- Host on which the process is running
- Process ID

Message Header

Message headers generated by CodeWizard include:

- Item violated and severity of violation
- File name
- Line number of violation

All messages sent to Insra are marked with a special icon. Please refer to the following table for a brief description of each icon.

Icon	Explanation
	CodeWizard violation message
	Trapped Signal

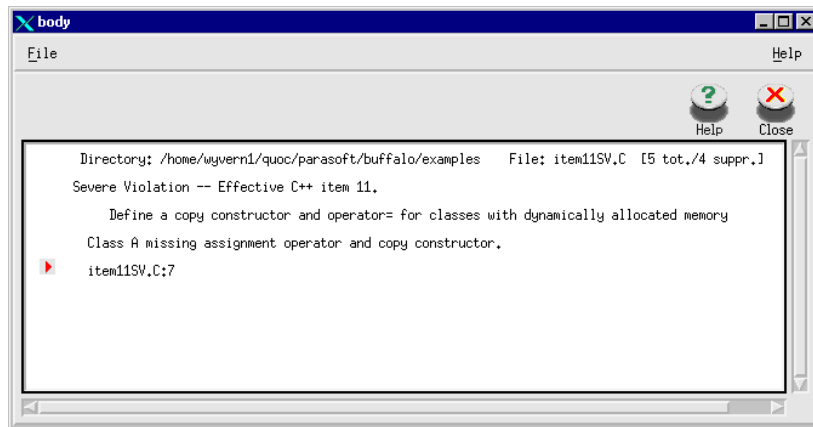
Clicking the ‘+’ sign next to the message header will reveal the source code where the violation occurs. Double-clicking a message header brings up a Message window.

Error Message Window

The Message window opens when you double-click a message header. This window contains the violation message for the selected message header.

CodeWizard's Message window contains:

- The line of source code where the violation occurred.
- An explanation of the violation detected.



To make immediate corrections to the source code, simply double-click the excerpt of code shown in Insra, or click the arrow in the Message window. This will open the file in a text editor, and bring you directly to the line containing the violation.

Sending Messages to Insra

By default, both CodeWizard sends output messages to Insra.

You may choose to send your output to another location, such as `stderr`, `stdout`, or a report file of your choice.

To change your output location, you would enter the following line to your `.psrc` file:

```
codewizard.report_file <filename>
```

The option

```
codewizard.runtime.report_file insra
```

will send only runtime messages to Insra. Compile-time messages will be sent to `stderr`.

The option

```
codewizard.compile.report_file insra
```

will send only compile-time messages to Insra. Run-time messages will be sent to `stderr`.

To direct messages back to Insra, simply replace the above line with the following line in your `.psrc` file:

```
codewizard.report_file insra
```

Your compile-time and run-time messages will be sent to Insra.

When CodeWizard is configured to send output to Insra, CodeWizard attempts to establish a connection to Insra each time a violation is detected. If Insra is not yet running, it will automatically start. Once the connection is established, a session header and all corresponding message headers will be reported in the order they were detected. Each new compilation or program, with its own session header and messages, will be displayed in the order in which it connected to Insra.

Viewing and Navigating

Message headers sent to Insra are denoted by specific icon. (See “The Insra Display”). The body of the currently selected message is displayed in a separate Message window. Double-click the message header to view the message itself. The message header area and the Message window are both resizable, and scroll bars are also available to access text that is not visible.

Currently active messages become inactive when they are deleted or suppressed

Deleting Messages

Once violation messages have been read and analyzed, you may wish to clear them from the window. The **Delete** button on the Insra tool bar allows you to remove messages from the display as violations are corrected in your code. A message or an entire session may be removed by selecting the corresponding entry in the message header area and clicking the **Delete** button. A message can also be deleted by selecting **Mes-sages> Delete** from the menu bar.

Suppressing Messages

You can use suppressions to tailor CodeWizard's output to your development team and the current project. By default, CodeWizard is configured to suppress all violations that have an Informational severity and all violations in the Universal Coding Standard category, then unsuppress any violations that have a Severe Violation severity and unsuppress any enabled User items.

Suppressions are entered in the Suppressions Control Panel, which is accessible by clicking the **Suppress** button in Insra when Insra is running and contains violations.

For more information about suppressing and unsuppressing messages, see "Customizing Output With Suppressions".

The Suppressions Tool Bar

Moving from left to right across the suppressions tool bar:

The **Prev** and **Next** arrows select the next higher or next lower suppression option, respectively.

The **Up** and **Down** arrows move the currently selected suppression option up or down in the order of options. Because CodeWizard follows suppression options in the order they are listed (from top to bottom), they order in which they are listed affects the outcome of the suppressions.

The **Delete** button deletes the currently selected suppression option.

The **Insert** button inserts a new suppression option below the currently selected option. If you had a violation message selected when you pressed the **Suppress** button on the Insra GUI, a suppression option for that particular violation will be inserted. Otherwise, the default suppression option (suppress all violation messages) will be inserted.

The **Save** button writes all the suppression options which have been marked as persistent (see below) into the indicated `.psrc` files (see below). These suppression options will be in effect the next time you use CodeWizard.

The **Help** button provides context-sensitive help, which in this window means that clicking anywhere will bring up this file.

The **Close** button closes the suppression window.

Miscellaneous Issues

Suppression options are read from top to bottom when determining which messages will be displayed and which will not.

Kill Process

When an active connection is selected, pressing the **Kill** button will stop the selected compilation or execution.

Viewing Source Files

You can view the corresponding source file and line number for a particular violation message by double clicking on any line of the stack trace displayed in the Message window. In most cases, the file and line number associated with a given message have been transmitted to Insra. If Insra is unable to locate the source file, a dialog box will appear requesting that you indicate the correct source file.

Selecting an Editor

In addition to the location of the source file, Insra must also know the name of your editor, and (if available) the command line syntax, in order to display the correct file and line from the original source code.

Insra obtains information by reading the `.psrc` option

```
insra.visual [editor_command]
```

This command may contain the special token `%f` and `%l`, which represent the file name and line number, respectively.

The command will then be executed to lead the file into your editor. It is most important to include the full path of any binary that lives in a location not pointed to by your `PATH` environment variable. If the variable has not been set, `vi` will be used by default.

Some editors are not X applications and must be run in a terminal window. `vi` requires the following command in order to lead the file successfully:

```
insra.visual xterm -e vi +%l %f
```

Other editors, e.g. Emacs, do not require an external terminal program like `xterm` when configured for use as an X application. In this case, the command string should be similar to the following:

```
insra.visual emacs +%l %f
```

Note: Most implementations of `vi` and Emacs appear to be sensitive to the order of the line number and file name command line arguments, requiring the line number to precede the file name.

Saving/Loading Messages

All current messages can be saved to a file by selecting **File> Save** or **File> Save As** from the menu bar. A dialog box allows you to select the destination directory and name of the report file. Report files have the default extension `rpt`. After a report file name has been selected, subsequent **File> Save** selections save all current messages into the report file without prompting for a new filename. A previously saved report file can

be loaded by selecting **File> Load** from the menu bar. A dialog box then allows you to select which report file to load.

Help

Online help can be obtained by choosing **Help** from the menu bar. This will provide a list of topics on the use of Insra.

Context-sensitive help is accessible by simply clicking the **Help** button on the tool bar. When selected, the mouse cursor changes to the question mark arrow combination; clicking any visual element of Insra will open up a help window with a description of that item.

Setting Preferences

You can modify Insra's appearance with `.psrc` configuration options.

These options are:

`insra.body_background_color [color]`

Specifies the color used for the message body area background. The default is white.

`insra.body_font [font]`

Specifies the font used for the message body text. The default is fixed.

`insra.body_height [number of rows]`

Specifies the starting height of the Message window in number of rows of visible text. The default is 8.

`insra.body_text_color [color]`

Specifies the color used for the message body text. The default is black.

`insra.body_width [columns of text]`

Specifies the starting width of the Message window in number of columns of visible text. The default is 80, but if this value is set to a differ-

ent value than `header_width`, then the larger value will be used.

`insra.header_background_color [color]`

Specifies the color used for the message header area background. The default color is white.

`insra.header_font [font]`

Specifies the font used for the message header text. The default is fixed.

`insra.header_height [number of rows]`

Specifies the starting height of the message header in number of rows of visible text. The default is 8.

`insra.header_highlight_color [color]`

Specifies the color used to indicate the currently selected message or session header in the message header area. The default is `LightSteelBlue2`.

`insra.header_highlight_text_color [color]`

Specifies the color used for the text of the currently selected message of session header in the message header area. The default is black.

`insra.header_session_color [color]`

Specifies the color used for session header text. The default is `LightSkyBlue3`.

`insra.header_session_text_color [color]`

Specifies the color used for session header text. The default is black.

`insra.header_text_color [color]`

Specifies the color used for message header text. The default color is black.

`insra.header_width [number of columns]`

Specifies the starting width of the header area in number of columns

of visible text. The default is 80, but if this value is set to a different value than `body_width`, the larger value will be used.

`insra.port [port_number]`

Specifies which port Insra should use to communicate with Insure++ compiled programs. The default is 3255.

`insra.toolbar [on or off]`

Specifies whether Insra's tool bar is displayed. All tool bar commands can be chosen from the menu bar. The default is on.

`insra.visual [editor command]`

Specifies how Insra should call an editor to display the line of source code causing the violation. Insra will match the `%l` token to the line number and the `%f` token to the file name before executing the command. Setting this option with no command string disables source browsing from Insra. The default is

```
xterm -e vi +%l %f
```

Troubleshooting

Insra Does Not Start Automatically

Symptom:

While compiling or running, your program seems to hang when output is directed to Insra and Insra is not yet running.

Solution:

Run Insra by hand. Type

```
insra &
```

at the prompt, wait for the Insra window to appear, and then run or compile your program again. Output should now be sent to Insra.

Multiple Users of Insra on One Machine

Symptom:

When more than one user is attempting to send message reports to Insra, messages are lost.

Solution:

Each invocation of Insra requires a unique port number. By default, Insra uses port 3255. If collisions are experienced, e.g. multiple users are on one machine, set the `.psrc` option `insra.port` to a different port above 1024. Ports less than 1024 are officially reserved for suid-root programs and should not be used with Insra.

Source Browsing is Not Working

Symptom:

```
***Error while attempting to spawn browser  
execvp failed!
```

Solution:

Insra attempted to launch your editor to view the selected source file, but could not locate either xterm or your editor on your path. Please make sure that both of these applications are in directories that are on your path or that you call them with their complete pathnames.

Items Enforced

CodeWizard reports violations of "items" described in several popular Scott Meyers books, including *Effective C++* and *More Effective C++*, and in an article written by Martin Klaus and published in the February 1997 issue of *Dr. Dobbs' Journal*. Additional items, referred to as Universal Coding Standards, and new items reported by Meyers and Martin Klaus are also included.

Using CodeWizard's new RuleWizard feature, you can even create your own "custom" items for CodeWizard to enforce.

Each item, along with a general description, is accompanied by its category, the severity of the violation, and whether it is enabled by default.

Items reported by CodeWizard have different levels of severity. The higher the level of severity an item is assigned, the greater the chance that violating that error will result in a bug. Each level is identified by its own unique abbreviation.

The following is a table of abbreviations used in this section and by CodeWizard.

Severity	Explanation
I	Informational: least chance of resulting in a bug
PV	Possible Violation
V	Violation
PSV	Possible Severe Violation
SV	Severe Violation: greatest chance of resulting in a bug

CodeWizard items are labeled "ecpp-<item number>" for *Effective C++* violations, "mecpp-<item number>" for *More Effective C++* violations, "MK-<item number>" for Meyers-Klaus violations, "ucs-<item number>" for ucs violations, and "user-<item number>" for custom coding standards created in RuleWizard, as well as built-in, modifiable coding standards.

Each item is described in depth within the appropriate section:

- Effective C++ Items
- More Effective C++ Items
- Meyers-Klaus Items
- Universal Coding Standard Items
- User Items

Effective C++ Items

Category	Item	Description	Violation	Enabled
Shifting from C to C++	2	Prefer <code>iostream.h</code> to <code>stdio.h</code> .	I	N
	3	Use <code>new</code> and <code>delete</code> instead of <code>malloc</code> and <code>free</code> .	V	Y
Memory Management	5	Use the same form in corresponding calls to <code>new</code> and <code>delete</code> .	V	Y
	6	Call <code>delete</code> on pointer members in destructors.	I / V	N / Y
	7	Check the return value of <code>new</code> .	I	N
	8	Adhere to convention when writing <code>new</code> .	V	Y
	9	Avoid hiding the global <code>new</code> .	SV	Y
	10	Write <code>delete</code> if you write <code>new</code> .	SV	Y

Constructors, Destructors, and Assignment Operators	11	Define a copy constructor and assignment operator for classes with dynamically allocated memory.	V/ SV	Y / Y
	12	Prefer initialization to assignment in constructors.	I	N
	13	List members in an initialization list in the order in which they are declared.	V	Y
	14	Make destructors virtual in base classes.	SV	Y
	15	Have operator= return a reference to this*.	SV	Y
	16	Assign to all data members in operator=.	PSV	Y
	17	Check for assignment to self in operator=.	PV	Y

Classes and Functions: Design and Declaration	19	Differentiate among member functions, global functions, and friend functions.	V	Y
	20	Avoid data members in the public interface.	V	Y
	22	Pass and return objects by reference instead of by value.	V	Y
	23	Don't try to return a reference when you must return an object.	PSV	Y
	25	Avoid overloading on a pointer and a numerical type.	V	Y

Classes and Functions: Implementation	29	Avoid returning “handles” to internal data from const member functions.	SV	Y
	30	Avoid member functions that return pointers or references to members less accessible than themselves.	V	Y
	31	Never return a reference to a local object or a dereferenced pointer initialized by new within the function.	PSV / SV	Y / Y
Inheritance and Object-Oriented	37	Never redefine an inherited nonvirtual function.	PV/V	Y / Y
	38	Never redefine an inherited default parameter value.	I / V	N / Y
	39	Avoid casts down the inheritance hierarchy.	I	N

Item ecpp_2

Prefer `iostream.h` to `stdio.h`

CodeWizard finds instances of `stdio.h` functions (such as `scanf/printf`) and suggests changing them to `iostream.h` functions (such as `operator>>` and `operator<<`). Errors are marked as violations of Item `ecpp_02`.

Reason for rule: `iostream.h` functions are type-safe and extensible.

Note: This item is suppressed by default.

Example

```
/*
 * Item 2 - Prefer iostream.h to stdio.h
 */
#include <stdio.h>

int main()
{
    printf("%s\n", "Hello World");    // ECPP item 2 viola-
tion
    return 0;
}
```

CodeWizard Output

```
[item02.C:8] Prefer iostream.h to stdio.h
Informational: Effective C++ item 2
```

Item ecpp_3

Use new and delete instead of malloc and free

CodeWizard will warn you if you use `malloc` and `free` in your code. Errors are marked as violations of Item ecpp_3.

Reason for rule: `new` and `delete` can handle constructors and destructors.

Example

```
/*
 * Item 3 - Use new and delete instead of malloc and free
 */
#include <malloc.h>
int main()
{
    char *pc;
    pc = (char *)malloc(100);    // ECPP item 3 violation
    free(pc);                   // ECPP item 3 violation

    return 0;
}
```

CodeWizard Output

```
[item03.C:10] Use new instead of malloc
Violation: Effective C++ item 3
```

```
[item03.C:11] Use delete instead of free
Violation: Effective C++ Item 3
```

Item ecpp_5

Use the same form in corresponding calls to new and delete

CodeWizard checks calls to `new` and `delete` to make sure that they use the same form; it will report a violation if you call `new` but forget to use `[]` when calling `delete`. Errors are marked as violations of Item ecpp_5.

Reason for rule: If you do not use the same form in corresponding calls to `new` and `delete`, an incorrect number of destructors may be called.

Example

```
/*
 * Item 5 - Use the same form in corresponding
 * calls to new and delete
 */
class A
{
public:
    A() {}
};

int main()
{
    A *a = new A[100];

    delete a;                // ECPP item 5 violation
}
```

CodeWizard Output

```
[item05.C:15] Use the same form in corresponding
```

```
calls to new and delete
Violation: Effective C++ item 5
Found new[] with delete.
```

Item ecpp_6

Call delete on pointer members in destructors

CodeWizard warns you if it finds a pointer member that has no corresponding `delete` in the destructor. Errors are marked as violations of Item ecpp_6.

Reason for rule: Calling `delete` on pointer members in destructors prevents memory leaks now and as the code evolves in the future.

CodeWizard will report this item as either an Informational (I) or Violation (V) violation.

Example

```
/*
 * Item 6 - Call delete on pointer members in destructors.
 (SV,V)
 */
class A
{
public:
    A() {
        cptr1=new char;
        cptr2=new char;
    }
    ~A() { // ECPP item 6 violation
        delete cptr1;
        delete vptrl;
    }

private:
    char *cptr1;
```



```
void *vptr1;  
char *cptr2;  
};  
  
int main()  
{  
    return 0;  
}
```

CodeWizard Output

[item06.C:5] Call delete on pointer members in destructors

Violation: Effective C++ item 6

Class A contains pointer members not obviously deleted in destructor:

cptr2 ** allocated in constructor **

Item ecpp_7

Check the return value of new

CodeWizard will warn you if you do not check the return value of `new`. Errors are marked as violations of Item ecpp_7.

Reason for rule: In cases where `new` can't allocate the requested memory, it will return 0.

Note: This item is suppressed by default.

Example

```
/*
 * Item 7 - Check the return value of new
 */

int main()
{
    char *pc;

    pc = new char[10*10*10];           // ECPP item 7 violation
    pc[0] = 'x';
    delete [] pc;
    return 0;
}
```

CodeWizard Output

```
[item07.C:9] Check the return value of new
Informational: Effective C++ item 7
```

Item ecpp_8

Adhere to convention when writing `new`

When you write operator `new`, CodeWizard checks to make sure that you are being consistent with the default `new`. Errors are marked as violations of Item ecpp_8.

Reason for rule: If you do not adhere to convention when you write `new`, you may cause confusing inconsistencies for users of your `new` and `delete` operators.

Example

```
/*
 * Item 8 - Adhere to convention when writing new
 */
#include <stdlib.h>

class A
{
public:
    A() {}
    int* operator new(size_t size) // ECPP item 8 violation
        return (int*)(new int);
};

int main()
{
    return 0;
}
```

CodeWizard Output

```
[item08.C:10] New should return void *
```

```
Violation: Effective C++ item 8
```

Item ecpp_9

Avoid hiding the global new

CodeWizard can detect when the global `new` is hidden by an operator `new`. Errors are marked as violations of Item ecpp_9.

Reason for rule: If you hide the global `new`, normal `new` operator functionality will be unavailable to maintainers of your code.

Example

```
/*
 * Item 9 - Avoid hiding the global new
 */
#include <stdlib.h>
void ErrorHandler() {};
class A
{
public:
    void* operator new(size_t size, void (*pehf())) // ECPP
item 9 violation
    {
        return new int[size];
    }

    void operator delete(void *A)
    {
        delete A;
    }
};
int main()
{
    A *a = new (ErrorHandler) A;
```

```
    return 0;  
}
```

CodeWizard Output

```
[item09.C:12] Avoid hiding the global new  
Severe violation: Effective C++ item 9
```

Item ecpp_10

Write delete if you write new

CodeWizard makes sure that if you write your own version of `operator new`, you also write a corresponding `operator delete`. Errors are marked as violations of Item ecpp_10.

Reason for rule: If you write `delete` when you write `new`, you'll ensure that `new` and `delete` share the same assumptions.

Note: Item user_212 is a customizable version of this rule.

Example

```
/*
 * Item 10 - Write delete if you write new.
 */

#include <stdlib.h>
class A
{
    // ECPP item 10 violation
public:
    A() {}
    void* operator new(size_t size)
    {
        return new int[size];
    }
};
int main()
{
    A *a = new A;

    return 0;
}
```

CodeWizard Output

```
[item10.C:11] Write delete if you write new  
Severe violation: Effective C++ item 10
```


Item ecpp_11

Define a copy constructor and assignment operator for classes with dynamically allocated memory

CodeWizard will make sure copy constructors and assignment operators have been defined in classes with dynamically allocated memory. Errors are marked as violations of Item ecpp_11.

Reason for rule: By defining a copy constructor and assignment operator, you will achieve significant memory savings and increased speed.

Example

CodeWizard will report violations of Item 11 as either Violation (V) or Severe (SV). Examples of both are provided here.

```
/*
 * Item 11 - Define a copy constructor and an assignment
operator for classes
 *           with dynamically allocated memory. (V)
 */

class A
{
public:
    class X {}
    A() {                                     // ECPP item 11 violation
        cptr1 = new char[100];
        if (!cptr1) {
            throw X();
        }
    }
}
```

```

    }
    ~A() {}

private:
    char *cptr1;
    void *vptr1;
    char *cptr2;
};

int main()
{
    return 0;
}

/*
 * Item 11 - Define a copy constructor and an assignment
operator for classes
 *           with dynamically allocated memory. (SV)
 */

class A
{
private:
    char *cptr1;
    void *vptr1;
    char *cptr2;

public:
    A() {}
    ~A() {                                     // ECPP item 11 violation
        delete cptr1;
        delete vptr1;
        delete cptr2;
    }
};

```

```
    }  
};  
  
int main()  
{  
    return 0;  
}
```

CodeWizard Output

[item11V.C:7] Define a copy constructor and operator= for classes with dynamically allocated memory
Violation: Effective C++ item 11
Class A missing assignment operator and copy constructor.

[item11SV.C:7] Define a copy constructor and operator= for classes with dynamically allocated memory
Severe violation: Effective C++ item 11
Class A missing assignment operator and copy constructor.

Item ecpp_12

Prefer initialization to assignment in constructors

CodeWizard checks constructors to see if you are assigning data members when you should be initializing them. `const` and `reference` can only be initialized, never assigned. Errors are marked as violations of Item ecpp_12.

Reason for rule: If you use initialization instead of assignment, you will increase efficiency and call fewer member functions.

Exception: The only time you should use assignment instead of initialization for data members in a class is when you have a large number of data members of built-in types and you want them all to be initialized in the same way in each constructor.

Note: This item is suppressed by default.

Example

```
/*
 * Item 12 - Prefer initialization to assignment in constructors
 */

class A
{
public:
    A(int i, float j) {                // ECPP item 12 violation
        _idata = i;
        _fdata = j;
    }

private:

```

```
    int _idata;  
    float _fdata;  
};  
  
int main()  
{  
    return 0;  
}
```

CodeWizard Output

```
[item12.C:9] Prefer initialization to assignment  
in constructors  
Informational: Effective C++ item 12
```

```
[item12.C:10] Prefer initialization to assignment  
in constructors  
Informational: Effective C++ item 12
```

Item ecpp_13

List members in an initialization list in the order in which they are declared

CodeWizard checks initialization lists to make sure that members are listed in the same order there as they were when declared in the class. Errors are marked as violations of Item ecpp_13.

Reason for rule: Class members are initialized in the order of their declaration in the class, not by the order in which they are listed in a member initialization list.

Example

```
/*
 * Item 13 - List members in an initialization list in the
 *           order in which they are declared
 */

class A
{
public:
    // ECPP item 13 violation
    A(int i1, int i2, int i3) : idata3(i3), idata2(i2),
    idata1(i1) {}

private:
    int idata1, idata2, idata3;
};

int main()
{
    return 0;
}
```

}

CodeWizard Output

[item13.C:10] List members in an initialization list
in the order in which they are declared
Violation: Effective C++ item 13

Item ecpp_14

Make destructors virtual in base classes

CodeWizard will verify that destructors in base classes are virtual. Errors are marked as violations of Item ecpp_14.

Reason for rule: Declaring the destructor virtual tells the compiler that it must examine the object being deleted to see where to start calling destructors.

Example

```
/*
 * Item 14 - Make destructors virtual in base classes
 */

class Base
{
public:
    Base() {}
    ~Base() {} // ECPP item 14 violation
};

class Derived: public Base
{
public:
    Derived() {}
    ~Derived() {}
};

int main()
```



```
{  
    return 0;  
}
```

CodeWizard Output

[item14.C:6] Make destructors virtual in base classes
Severe violation: Effective C++ item 14
Class Base is a base class but does not have a virtual
destructor

Item ecpp_15

Have operator= return a reference to *this

CodeWizard makes sure your assignment operators return a reference to their left-hand argument, `*this`. Errors are marked as violations of Item ecpp_15.

Reason for rule: Having `operator=` return a reference to `*this` protects you from not knowing where the temporary gets destroyed and allows you to declare the `operator=`'s parameter as a reference to `const`, which is safer than just declaring it to be a reference.

Example

```
/*
 * Item 15 - Have operator= return a reference to *this
 */

class A
{
public:
    explicit A(int i = 0) : _i(i) {}
    void operator=(const A& a) // ECPP item 15 violation
    {
        if (&a == this) {
            return;
        }

        int _i = a._i;
        return;
    }
}
```

```
private:
    int _i;
};

int main()
{
    return 0;
}
```

CodeWizard Output

```
[item15.C:9] Have operator= return a reference to
*this
Severe violation: Effective C++ item 15
```

Item ecpp_16

Assign to all data members in operator=

If you write `operator=`, CodeWizard makes sure you assigned to every data member of your object. Errors are marked as violations of Item ecpp_16.

Reason for rule: Assigning to all data members in `operator=` allows you to take control of your assignments.

Example

```
/*
 * Item 16 - Assign to all data members in operator=
 */

class A
{
public:
    A() {}
    A& operator=(const A&);

private:
    int _x, _y, _z;
};

A& A::operator=(const A& a) // ECPP item 16 violation
{
    if (&a == this) {
        return *this;
    }
}
```

```
        _x = a._x;  
        _y = a._y;  
  
        return *this;  
    }  
  
int main()  
{  
    return 0;  
}
```

CodeWizard Output

[item16.C:16] Assign to all data members in operator=
Possible severe violation: Effective C++ item 16
Members not assigned:
_z

Item ecpp_17

Check for assignment to self in operator=

CodeWizard checks your code for aliasing in assignment operators. Errors are marked as violations of Item ecpp_17.

Reason for rule: Not checking for assignment to self in `operator=` can free resources that might be needed during the process of allocating new resources. Checking for assignment to self may also save you a lot of work that you would otherwise have to do to implement assignments.

Example

```
/*
 * Item 17 - Check for assignment to self in operator=
 */

class A{
public:
    A() {}
    A& operator=(A& a)                // ECPP item 17 violation
    {
        _i = a._i;
        return *this;
    }

private:
    int _i;
};

int main()
{
```

```
        return 0;  
    }
```

CodeWizard Output

[item17.C:8] Check for assignment to self in operator=
Possible violation: Effective C++ item 17
expected: if (&a == this) return *this;

Item ecpp_19

Differentiate among member functions, global functions, and friend functions

CodeWizard checks declared functions to make sure they are proper for the class type used. Disallowed functions are marked as violations of Item ecpp_19.

Reason for rule: Differentiating among functions leads to more object-oriented code with behavior that is more intuitive to use and maintain.

Example

```
/*
 * Item 19 - Differentiate among member, global, and friend
 functions.
 */

class Rational
{
public:
    Rational(int _num = 0, int _den = 1);
    int numerator() const { return _num; }
    int denominator() const { return _den; }

    Rational operator*(const Rational& rhs) const;    // ECPP
item 19 violation

private:
    int _num;
    int _den;
```



```
};

Rational Rational::operator*(const Rational& rhs) const
{
    Rational temp(_num * rhs.numerator(),
                  _den * rhs.denominator());

    return temp;
}

int main()
{
    return 0;
}
```

CodeWizard Output

```
[item19.C:12] Differentiate among member functions,
global functions and friend functions
Violation: Effective C++ item 19
Function * should be global
```

Item ecpp_20

Avoid data members in the public interface

CodeWizard warns you if data members are present in the public interface. Instead of making data members public, you should use functions. Errors are marked as violations of Item ecpp_20.

Reason for rule: Avoiding data members in the public interface provides more precise control over accessibility of data members, reduces confusion for callers, and enables functional abstraction (allowing the replacement of data members with computations without impacting users of the class). It also improves consistency, flexibility, and access-control.

Example

```
/*
 * Item 20 - Avoid data members in the public interface
 */

class A
{
public:
    int _idata2;           // ECPP item 20 violation

    int ReadData1(void) { return _idata1; }

    void WriteData1(int ival) { _idata1 = ival; }

private:
    int _idata1;
};
```

```
int main()
{
    return 0;
}
```

CodeWizard Output

```
[item20.C:6] Avoid data members in the public
interface
Violation: Effective C++ item 20
Public data members for class A:
_idata2
```

Item ecpp_22

Pass and return objects by reference instead of by value

CodeWizard can detect where an object has been passed by value instead of by reference. Code Wizard marks these errors as violations of Item ecpp_22.

Reason for rule: Passing and returning objects by reference is more efficient than passing by value because no new objects are being created and because it avoids the "slicing problem."

Exception: There are situations in which you can't pass an object by reference (see Item ecpp_23). There also may be a situation where the object is so small that it would be more efficient to pass by value instead of by reference.

Example

```
/*
 * Item 22 - Pass and return objects by reference instead
 *           of by value
 */

class A
{
public:
    A() {}
    A foo(const A a)      // 2 ECPP item 22 violations
    {                    //    passing variables into foo
by value
        return a;        //    returning by value when it
is not required
    }
```

```
};

int main()
{
    return 0;
}
```

CodeWizard Output

[item22.C:10] Pass objects by reference instead of
by value

Violation: Effective C++ item 22

Parameter a of function foo passed by value

[item22.C:10] Return objects by reference instead of by
value

Violation: Effective C++ item 22

Function foo returns object by value

Item ecpp_23

Don't try to return a reference when you must return an object

CodeWizard can detect when you return a reference where you are supposed to return an object. Errors are marked as violations of Item ecpp_23.

Reason for rule: Returning a reference instead of an object will result in corrupt data or a memory leak.

Example

```
/*
 * Item 23 - Don't try to return a reference when you must
 * return an object.
 */

class A
{
public:
    A(int num = 0, int den = 1) : _num(num), _den(den) {}
    int numerator() const { return _num; }
    int denominator() const { return _den; }

private:
    int _num;
    int _den;
};

// ECPP item 23 violation
A& operator*(const A& lhs, const A& rhs)
```

```
{
    A temp(lhs.numerator() * rhs.numerator(),
           lhs.denominator() * rhs.denominator());
    return temp;
}

int main()
{
    return 0;
}
```

CodeWizard Output

[item23.C:19] Don't try to return a reference when you must return an object
Possible severe violation: Effective C++ item 23
Operator * should return an object

Item ecpp_25

Avoid overloading on a pointer and a numerical type

CodeWizard can detect when you are overloading on a pointer and a numerical type. Errors are marked as violations of Item ecpp_25.

Reason for rule: Calling with an argument of zero will invoke the numerical type even though it is intuitively ambiguous.

Example

```
/*
 * Item 25 - Avoid overloading on a pointer and a numerical
 * type.
 */

class A
{
public:
    int func(char *ch)
    {
        char c=*ch;
        return 0;
    }
    int func(int i)           // ECPP item 25 violation
    {
        int var=i;
        return 0;
    }
};

int main()
```



```
{  
    return 0;  
}
```

CodeWizard Output

[item25.C:8] Avoid overloading on a pointer and a numerical type

Violation: Effective C++ item 25

Function func overloaded at:

[item25.C:8]

[item25.C:13]

Item ecpp_29

Avoid returning "handles" to internal data from const member functions

CodeWizard can detect when `const` member functions return “handles” to information that should be hidden. Errors are marked as violations of Item ecpp_29.

Reason for rule: If you return “handles” to internal data, you will allow calling functions to modify variables that they shouldn't have access to.

Example

```
/*
 * Item 29 - Avoid returning "handles" to internal data from
 *           const member functions
 */

class A
{
public:
    A()
    {
        ptr = 0;
    }
    operator char*() const { return ptr; }    // ECPP item
29 violation

private:
    char *ptr;
};

int main()
```

```
{  
    return 0;  
}
```

CodeWizard Output

[item29.C:13] Avoid returning "handles" to internal
data from const member functions
Severe violation: Effective C++ item 29

Item ecpp_30

Avoid member functions that return pointers or references to members less accessible than themselves

CodeWizard can detect when you write member functions that give clients access to restricted members. Errors are marked as violations of Item ecpp_30.

Reason for rule: Members are made private or protected to restrict access to them, so it doesn't make sense to write functions that give random clients the ability to freely access restricted members.

Exception: Cases where performance constraints dictate that you write a member function that returns a reference or pointer to a less-accessible member. To avoid sacrificing private and protected access restrictions, return a pointer to or a reference to a `const` object (see Item ecpp_21).

Example

```
/*
 * Item 30 - Avoid member functions that return pointers or
 *           references to members less accessible than them-
selves
 */

class A
{
public:
    int& GetA() { return _personalA; };        // ECPP item 30
violation

protected:
    int _personalA;
```

```
};

int main()
{
    return 0;
}
```

CodeWizard Output

[item30.C:9] Avoid member functions that return pointers or references to members less accessible than themselves

Violation: Effective C++ item 30

Item ecpp_31

Never return a reference to a local object or a dereferenced pointer initialized by new within the function

CodeWizard can detect when you return a reference to a local object or dereferenced pointer. Errors are marked as violations of Item ecpp_31.

Reason for rule: Returning a reference to a local object or a dereferenced pointer initialized by `new` within the function may cause a memory leak.

Example

CodeWizard will report violations of Item 31 as either Potentially Severe Violations (PSV) or Severe Violations (SV). Examples of both are provided here.

```
/*
 * Item 31 - Never return a reference to a local object or a
 *           dereferenced pointer initialized by new within
 *           the function
 */

class A
{
public:
    A(int xval, int yval) : _x(xval), _y(yval) {}

    friend A& operator+(const A& p1, const A& p2);

private:
    int _x, _y;
```

```

};

A& operator+(const A& p1, const A& p2)           // ECPP item
31 violation
{
    A *result = new A(p1._x + p2._x, p1._y + p2._y);

    return *result;
}

int main()
{
    return 0;
}

/*
 * Item 31 - Never return a reference to a local object or a
 *            dereferenced pointer initialized by new within
 *            the function
 */

class A
{
public:
    A(int xval, int yval) : _x(xval), _y(yval) {}

    friend A& operator+(const A& p1, const A& p2);

private:
    int _x, _y;
};

```

```

A& operator+(const A& p1, const A& p2)      // ECPP item 31
violation
{
    A result(p1._x + p2._x, p1._y + p2._y);

    return result;
}

int main()
{
    return 0;
}

```

CodeWizard Output

[item31PS.C:22] Never return a dereferenced local pointer initialized by new in this function scope
Possible severe violation: Effective C++ item 31
Dereferenced local pointer result returned.

[item31SV.C:22] Never return a reference to a local object
Severe violation: Effective C++ item 31
Reference to object result returned.

Item ecpp_37

Never redefine an inherited nonvirtual function

CodeWizard can detect when you redefine an inherited nonvirtual function. These errors are violations of Item ecpp_37.

Reason for rule: Redefining an inherited nonvirtual function may cause objects to behave incorrectly.

Example

CodeWizard will report Item 37 errors as either a Possible Violation (PV) or a Violation (V). Examples of both are provided here.

```
/*
 * Item 37 - Never redefine an inherited nonvirtual function
 */

template<class T>
class Base
{
public:
    void foo(T tval);
};

template<class T>
class Derived: public Base<T>
{
public:
    void foo(int tval);           // ECPP item 37 violation
};
```

```

int main()
{
    return 0;
}

/*
 * Item 37 - Never redefine an inherited nonvirtual function
 */

class Base
{
public:
    void func(void) {};
};

class Derived: public Base
{
public:
    void func(void) {};    // ECPP item 37 violation
};

int main()
{
    return 0;
}

```

CodeWizard Output

```

[item37PV.C:16] Never redefine an inherited nonvirtual
function
Possible violation: Effective C++ item 37
Redefinition of function foo in class Derived

```

Function foo is inherited from class Base

[item37V.C:14] Never redefine an inherited nonvirtual function

Violation: Effective C++ item 37

Redefinition of function func in class Derived

Function func is inherited from class Base

Item ecpp_38

Never redefine an inherited default parameter value

CodeWizard can detect where you redefine an inherited virtual function with a default parameter value. Errors are marked as violations of Item ecpp_38.

Reason for rule: As illustrated in the first example below, inherited nonvirtual functions should never be redefined. As illustrated in the second example below, virtual functions are dynamically bound and default parameter values are statically bound.

Example

CodeWizard will report Item 38 errors as either Informational (I) or as a Violation (V). Examples of both are provided here.

```
/*
 * Item 38 - Never redefine an inherited default parameter
value
 */

template<class T>
class Base
{
public:
    virtual void func(T i = -1111) {}
};

template<class T>
class Derived: public Base<T>
{
public:
```

```

        virtual void func(int i = 11) {}// EECPP item 38 violation
    };

    int main()
    {
        return 0;
    }

    /*
     * Item 38 - Never redefine an inherited default parameter
     value
     */

    class Base
    {
    public:
        virtual int func(int i = -1111) { return i; }
    };

    class Derived: public Base
    {
    public:
        virtual int func(int i = 11) { return i; }    // ECPP
        item 38 violation
    };

    int main()
    {
        return 0;
    }

```

CodeWizard Output

[item38I.C:16] Never redefine an inherited default parameter value

Informational: Effective C++ item 38

Function func in class Derived redefines default parameter i

[item38V.C:14] Never redefine an inherited default parameter value

Violation: Effective C++ item 38

Function func in class Derived redefines default parameter i

Item ecpp_39

Avoid casts down the inheritance hierarchy

CodeWizard can detect casts from a base class pointer to a subclass pointer. Errors are marked as violations of Item ecpp_39.

Reason for rule: Allowing casts down the inheritance hierarchy leads to maintenance problems, and downcasting from a base class is always illegal.

Note: This item is suppressed by default.

Example

```
/*
 * Item 39 - Avoid casts down the inheritance hierarchy
 */

class Base {};

class Derived: public Base {};

int main()
{
    Base *pb;
    Derived *pd = (Derived *)pb; // ECPP item 39 violation

    return 0;
}
```

CodeWizard Output

```
[item39.C:12] Avoid casts down the inheritance
```

hierarchy

Informational: Effective C++ item 39

Class Derived inherits from class Base

Item ecpp_39

Items Enforced

More Effective C++ Items

Category	Item	Description	Violation	Enabled
Basics	2	Prefer C++-style casts.	V	Y
Operators	5	Be wary of user-defined conversion functions.	I / V	N / Y
	6	Distinguish between prefix and postfix forms of increment and decrement operators.	V	Y
	7	Never overload &&, , or ,.	V	Y
Efficiency	22	Consider using op= instead of stand-alone op.	V	Y
	24	Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI.	V	Y
Techniques	26	Limiting the number of objects of a class.	I	N

Item mecpp_02

Prefer C++ style casts

CodeWizard can detect C-style casts in your code. Errors are marked as violations of Item mecpp_02.

Reason for rule: C++ casts are more specific than C casts and are much easier to locate and read.

Example

```
//  
// More Effective C++ Item 2  
//  
  
class Base  
{  
public:  
    Base();  
    virtual ~Base();  
};  
  
class Derived : public Base  
{  
public:  
    Derived();  
    ~Derived();  
};  
  
int main()  
{  
    Base *pB;
```

```
Derived *pD = (Derived *) pB; // MECPP item 2 violation.  
return 0;  
}
```

CodeWizard Output

[ME-item02.C:23] Prefer C++-style casts

Violation: More Effective C++ item 2

Item mecpp_05

Be wary of user-defined conversion functions

CodeWizard warns you when it finds user-defined conversion functions. Errors are marked as violations of Item mecpp_ 05.

Reason for rule: Using user-defined conversion functions may result in incorrect program behavior.

Example

CodeWizard will report violations of Item 5 as either Information (I) or Violation (V). Examples of both are provided here.

```
//
// More Effective C++ Item 5
//

class A
{
public:
    A(int i): _i(i) {}           // MECPPE item 5 violation
    virtual ~A() {}

    operator int() const         // MECPPE item 5 violation
    {
        return _i;
    }

private:
    int _i;
```

```
};

int main()
{
    int j = 10;

    A a(12);

    if (a < j) {                // MECPP item 5 violation
        exit(1);
    }
    return 0;
}
```

CodeWizard Output

[ME-item05.C:8] Be wary of user-defined conversion functions
 Informational: More Effective C++ item 5
 Constructor allowing conversion should be made explicit

[ME-item05.C:11] Be wary of user-defined conversion functions
 Violation: More Effective C++ item 5
 User-defined conversion should be made explicit

[ME-item05.C:26] Be wary of user-defined conversion functions
 Violation: More Effective C++ item 5
 Use of implicit cast should be made explicit

Item mecpp_06

Distinguish between prefix and postfix forms of increment and decrement operators

CodeWizard warns you if you do not distinguish between prefix and postfix forms of increment and decrement operators. Failure to do so in ++ and -- operators results in a violation of Item mecpp_06.

Reason for rule: Prefix and postfix forms return different types (prefix forms return a reference and postfix forms return a const object). Postfix operators should be implemented in terms of the prefix operator.

Example

```
//
// More Effective C++ Item 6
//

class A
{
    friend bool operator==(const A&, int);

public:
    explicit A(int i = 0): _i(i) {}
    ~A() {}

    A& operator=(const A& a)
    {
        if (&a == this) {
            return *this;
        }
    }
}
```

```

        _i = a._i;
        return *this;
    }
    A& operator=(int i)
    {
        _i = i;
        return *this;
    }
    A& operator++()
    {
        ++_i;
        return *this;
    }
    const A operator++(int)
    {
        A temp = *this;
        ++_i;
        return temp;
    }

private:
    int _i;
};

bool operator==(const A& a, int i)
{
    return (a._i == i);
}

bool operator!=(const A& a, int i)
{
    return (!(a == i));
}

```



```
}

int main()
{
    A a;

    for (a = 0; a != 10; a++) { // MECPP item 6 violation
        // Do something
    }

    return 0;
}
```

CodeWizard Output

[ME-item06.C:57] Distinguish between prefix and postfix forms of increment and decrement
Violation: More Effective C++ item 6
Prefix form is recommended here

Item mecpp_07

Never overload &&, ||, or ,

CodeWizard can detect when you overload operator `&&` , `||` or `,`. Errors are marked as violations of item mecpp_07.

Reason for rule: Overloading these operators changes the way the compiler reads the semantics of an expression, resulting in unpredictable program behavior.

Example

```
//
// More Effective C++ Item 7
//

class A
{
public:
    A(int i) : _i(i) {}
    ~A();

    int value() { return _i; }

private:
    int _i;
};

operator&&(A& lhs, A& rhs) { // MECPP item 7 violation
    return lhs.value() && rhs.value();
}

int main()
```

```
{  
    return 0;  
}
```

CodeWizard Output

[ME-item07.C:17] Never overload &&, || or ,
Violation: More Effective C++ item 7
Function && is overloaded

Item mecpp_22

Consider using op= instead of stand-alone op

CodeWizard warns you whenever it finds stand-alone versions of operator. Errors are marked as violations of Item mecpp_22.

Reason for rule: Assignment versions of operator are more efficient than stand-alone versions.

Example

```
//
// More Effective C++ Item 22
//

class A
{
public:
    A(int i) : _i(i) {}
    ~A();

    int value() { return _i; }

    A operator+() {}           // MECPP item 22 violation

private:
    int _i;
};

int main()
{
```

```
    return 0;  
}
```

CodeWizard Output

[ME-item22.C:13] Consider using op= instead of
stand-alone op

Violation: More Effective C++ item 22

Operator + found, but no appropriate operator += found

Item mecpp_24

Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI

CodeWizard warns you when virtual functions, multiple-inheritance, virtual base classes, and RTTI will be implemented inefficiently. Errors are marked as violations of Item mecpp_ 24.

Reason for rule: Inefficient implementation may have noticeable effects on the size of objects and the speed at which member functions execute; this may significantly impact performance.

Example

```
//
// More Effective C++ Item 24
//

class A
{
public:
    A(int i) : _i(i) {}
    virtual ~A() {} // MECPP item 24 violation

    int value() { return _i; }

private:
    int _i;
};

int main()
```

```
{  
    return 0;  
}
```

CodeWizard Output

[ME-item24.C:9] Understand the costs of virtual functions ...

Violation: More Effective C++ item 24

Function ~A is both virtual and inline

Item mecpp_26

Limiting the number of objects in a class

CodeWizard can detect when you need to limit the number of objects in a class. Errors are marked as violations of Item mecpp_ 26.

Reason for rule: Limiting the number of objects in a class can reduce code complexity.

Note: This item is suppressed by default.

Example

```
//
// More Effective C++ Item 26
//

class Base
{
public:
    class X {};
    Base()
    {
        if(numObjects >= 1)
            throw X();

        // constructor stuff here

        ++numObjects;
    }

    ~Base()
```



```
        {
            --numObjects;

            // destructor stuff here
        }

private:
    static int numObjects;
};

int main()
{
    return 0;
}
```

CodeWizard Output

[item06.C:6] Limiting the number of objects of a class
Informational: More Effective C++ item 26

Meyers-Klaus Items

Category	Item	Description	Violation	Enabled
Constructors/ Destructors/ Assignment	13	Avoid calling virtual functions from constructors and destructors.	SV	Y
Implementation	23	Avoid using "..." in function parameter list.	I	N

Item MK_13

Avoid calling virtual functions from constructors and destructors

CodeWizard can detect when you call virtual functions from constructors and destructors. These errors are reported as violations of Item MK_13.

Example

```
//
// Meyers-Klaus Item 13
//

class Base
{
public:
    Base() { init(); }           // MK item 13 violation
    virtual ~Base();

protected:
    virtual void init();
    int _i;
};

void Base::init()
{
    _i = 10;
}

class Derived : public Base
```

```

{
public:
    Derived() { init(); }           // MK item 13 violation
    ~Derived();

private:
    void init();
};

void Derived::init()
{
    _i = 20;
}

int main()
{
    return 0;
}

```

CodeWizard Output

[MK-item13.C:8] Avoid calling virtual functions from constructors and destructors
 Severe violation: Meyers-Klaus item 13
 Virtual function init called.

[MK-item13.C:25] Avoid calling virtual functions from constructors and destructors
 Severe violation: Meyers-Klaus item 13
 Virtual function init called.

Item MK_23

Avoid using “...” in function parameter lists

CodeWizard can detect when you use “...” in function parameter lists. These errors are reported as violations of Item MK_23.

Note: This item is suppressed by default.

Example

```
//  
// Meyers-Klaus Item 23  
//  
  
class A  
{  
public:  
    A() { init(); }  
    virtual ~A();  
  
private:  
    void init(...);  
    int _i;  
};  
  
void A::init(...)  
{  
    _i = 10;  
}
```

```
int main()  
{  
    return 0;  
}
```

CodeWizard Output

```
[MK-item23.C:17] Avoid using "... " in function  
parameter list  
Informational: Meyers-Klaus item 23
```

Universal Coding Standard Items

Item	Description	Violation	Enabled
2	Do not declare protected data members.	V	No
3	Do not declare the constructor or destructor to be inline.	V	No
4	Declare at least one constructor to prevent the compiler from doing so.	V	No
5	Pointers to functions should use a typedef.	V	No
6	Never convert a const to a non-const.	SV	Yes
7	Do not use the ?: operator.	V	No
8	Each class must declare the public, protected, and private sections in that order.	V	No
9	In the public section entities shall be declared in the following order: Constructors, Destructors, Member functions, Member conversion functions, Enumerations, and other.	V	No

10	In the protected section entities shall be declared in the following order: Constructors, Destructors, Member functions, Member conversion functions, Enumerations, and others.	V	No
11	In the private section entities shall be declared in the following order: Constructors, Destructors, Member functions, Member conversion functions, Enumerations, and others.	V	No
12	If a function has no parameters, use () instead of (void).	V	No
13	If, else, while, and do statements shall be followed by a block, even if it is empty.	V	No
14	If a block is a single statement, enclose it in braces.	V	No
15	Whenever a global variable or function is used, use the :: operator.	I	No
16	Do not use public data members.	V	No
17	If a class has any virtual functions it shall have a virtual destructor.	SV	Yes
18	Public member functions shall return const handles to member data.	SV	Yes

19	A class that has pointer members shall have an operator= and a copy constructor.	V	No
20	If a subclass implements a virtual function, use the virtual keyword.	V	No
21	Member functions shall not be defined in the class definition.	V	No
22	Ellipses shall not be used.	V	No
23	Functions shall explicitly declare their return types.	V	No
24	A pointer to a class shall not be converted to a pointer of a second class unless it inherits from the second.	SV	Yes
25	A pointer to an abstract class shall not be converted to a pointer that inherits from that class.	SV	Yes
26	Do not use the friend mechanism.	V	No
27	When working with float or double values, use <= and >= instead of ==.	SV	Yes
28	Do not overload functions within a template class.	SV	Yes
29	Do not define structs that contain member functions.	V	No
30	Do not directly access global data from a constructor.	SV	Yes

31	Do not use multiple inheritance.	V	No
32	Initialize all variables.	V	No
33	All pointers should be initialized to zero.	V	No
34	Always terminate a case statement with break.	V	No
35	Always provide a default branch for switch statements.	V	No
36	Do not use the goto statement.	V	No
37	Provide only one return statement in a function.	I	No

Item ucs_02

Do not declare protected data members

Declaring protected variables in a class results in the variables becoming visible to derived classes. CodeWizard can detect if you declare protected data members. These errors are reported as violations of Item ucs_02.

Reason for rule: When you declare data members protected instead of private, you expose members to derived classes that could be encapsulated in member functions.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 02 - Do not declare protected data members
3:  * /
4: class A
5: {
6: protected:
7:     int i;                // UCS02 Violation
8: };
```

CodeWizard Output

```
[ucs02.C:6] Do not use protected data members
Violation: Universal Coding Standards item 2
Class A declares the following protected data members:
i
```

Item ucs_03

Do not declare the constructor or destructor to be inline

Inline functions that invoke other inline functions can be too complex for the compiler to make them inline. Because constructors always invoke the constructors of their base classes and member data, CodeWizard reports use of inline constructors and destructors as violations of Item ucs_03.

Reason for rule: Portability. Functions that invoke other inline functions often become too complex for the compiler to be able to make them inline, despite the fact that they appear to be small. This problem is especially common with constructors and destructors. A constructor always invokes the constructors of its base classes and member data before executing its own code. If you avoid making constructors and destructors inline, you will make the code more portable to compilers that get confused when generating complex nested inline functions.

Note: This item is suppressed by default.

Example

```

1: /*
2:  * UCS item 03 - Do not declare the constructor or
  destructor to be inline
3:  * /
4: class A
5: {
6: public:
7:     A {}                      // UCS03 Violation
8:     ~A {}                     // UCS03 Violation
9: };

```

CodeWizard Output

```
[ucs03.C:6] Do not define any constructor to be inline  
Violation: Universal Coding Standards item 3  
Class A defines an inline constructor
```

Item ucs_04

Declare at least one constructor to prevent the compiler from doing so

If you do not write at least one constructor in a class, the compiler will write a public constructor for you by default. CodeWizard can detect if you do not declare at least one constructor and report these errors as violations of Item ucs_04.

Reason for rule: Readability. If you follow this rule, you will make class initialization explicit and prevent the compiler from initializing members improperly--especially pointer members.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 04 - Declare at least one constructor to
3:  * prevent the compiler from doing so
4:  * /
5: class A
6: {
7: };                                // UCS04 Violation
```

CodeWizard Output

```
[ucs04.C:7] Declare at least one constructor to prevent the
compiler from doing so
```

```
Violation: Universal Coding Standards item 4
```

```
Class A does not define any constructors
```

Item ucs_05

Pointers to functions should use a typedef

Using a `typedef` when writing pointers to functions will result in more readable code that is easier to maintain and port. CodeWizard can detect when pointers to functions do not use a `typedef`. These errors are reported as violations of Item ucs_05.

Reason for rule: When you use a `typedef`, you provide greater readability for function pointers.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 05 -Pointers to functions should use a typedef
3:  * /
4: void (*foo) (int);          // UCS05 Violation
```

CodeWizard Output

```
[ucs05.C:7] Pointers to functions shall be implemented with
a typedef
```

```
Violation: Universal Coding Standards item 5
```

Item ucs_06

Never convert a const to a non-const

Using explicit type conversions to convert `const` member data to non-const will prevent the compiler from allocating constants in ROM. CodeWizard can detect if you convert a `const` to a `non-const`. These errors are reported as violations of Item ucs_06.

Reason for rule: Converting `const` to `non-const` can undermine the data integrity by allowing values to change that are assumed to be constant. This practice also reduces the readability of the code, since you cannot assume `const` variables to be constant.

Example

```

1: /*
2:  * UCS item 06 - Never convert a const to a non-const
3:  * /
4: int main()
5: {
6:     const int a = 10
7:     int b;
8:
9:     b = (int)a;           // UCS06 Violation
10: };

```

CodeWizard Output

```

[ucs06.C:10] Never convert a const to a non-const
Severe violation: Universal Coding Standards item 6

```


Item ucs_07

Do not use the ?: operator

Using the ?: operator can make your code more difficult for others to read and understand. CodeWizard can detect if you use the ?: operator. This error is reported as violations of Item ucs_07.

Reason for rule: The ?: operator is difficult to read and leads to code obfuscation.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 07 - Do not use the ?: operator
3:  * /
4: int main()
5: {
6:     return (1 ? 1 : 0); // UCS07 Violation
7: };
```

CodeWizard Output

```
[ucs07.C:7] Avoid using the ?: operator
Violation: Universal Coding Standards item 7
```

Item ucs_08

Each class must declare the public, protected, and private sections in that order

Using the above convention, everything of general interest to a user (the public section) is gathered in the beginning of the class definition while items of the least general interest (the private section) are gathered last. CodeWizard reports violations of this convention as violations of Item ucs_08.

Reason for rule: Readability.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 08 - Each class must declare the public,
3:  * protected, and private sections in that order
4:  * /
5: class A
6: {
7: public:
8:     );
9:
10: private:                                // UCS08 Violation
11:     void foo();
12:
13: protected:
14:     void bar();
```

```
15: };
```

CodeWizard Output

[ucs08.C:7] Each class must declare the public, protected, and private sections in that order

Violation: Universal Coding Standards item 8

class A declares these sections out of order.

Item ucs_09

In the public section, entities shall be declared in the following order: Constructors, Destructors, Member functions, Member conversion functions, Enumerations, and other

Following the above ordering convention will make it easier for those unfamiliar with a class to figure out its functionality. CodeWizard reports violations of this convention as violations of Item ucs_09.

Reason for rule: Readability.

Note: This item is suppressed by default.

Example

```

1: /*
2:  * UCS item 09 - In the public section entities shall be
   declared in the
3:  *   following order: Constructors, Destructors, Member
4:  *   functions, Member conversion functions,
5:  *   Enumerations, and other
6:  * /
7: class A
8: {
9: public:
10:     );
11:     int foo();           // UCS09 Violation
12:     );
13: };

```

CodeWizard Output

[ucs09.C:9] In the public section of a class items shall be declared in the following order: Constructors, Destructor, Member Functions, Member Operator Function, Enumerations, other

Violation: Universal Coding Standards item 9

Item ucs_10

In the protected section entities shall be declared in the following order: Constructors, Destructors, Member functions, Member conversion functions, Enumeration, and other

Following the above ordering convention will make it easier for those unfamiliar with a class to figure out its functionality. CodeWizard reports violations of this convention as violations of item ucs_10.

Reason for rule: Readability.

Note: This item is suppressed by default.

Example

```

1: /*
2:  * UCS item 10 - In the protected section entities shall
3:  * be declared in the following order: Constructors,
4:  * Destructors, Member functions, Member conversion
5:  * functions, Enumerations, and other
6:  * /
7: class A
8: {
9: protected:
10:     );
11:     int foo();                // UCS10 Violation
12:     );
13: };

```

CodeWizard Output

[ucs10.C:9] In the protected section of a class items shall be declared in the following order: Constructors, Destructor, Member Functions, Member Operator Function, Enumerations, other

Violation: Universal Coding Standards item 10

Item ucs_11

In the private section entities shall be declared in the following order: Constructors, Destructors, Member functions, Member conversion functions, Enumerations, and other

Following the above ordering convention will make it easier for those unfamiliar with a class to figure out its functionality. CodeWizard can detect if you declare entities in the private section in the wrong order. These errors are reported as violations of Item ucs_11.

Reason for rule: Readability.

Note: This item is suppressed by default.

Example

```

1: /*
2:  * UCS item 11 - In the private section entities shall
3:  * be declared in the following order: Constructors,
4:  * Destructors, Member functions, Member conversion
5:  * functions, Enumerations, and other
6:  * /
7: class A
8: {
9: private:
10:     );
11:     int foo();           // UCS11 Violation
12:     );
13: };

```


CodeWizard Output

[ucs11.C:9] In the private section of a class items shall be declared in the following order: Constructors, Destructor, Member Functions, Member Operator Function, Enumerations, other

Violation: Universal Coding Standards item 11

Item ucs_12

If a function has no parameters, use () instead of (void)

CodeWizard can detect if you use `(void)` instead of `()` if a function has no parameters. Maintaining this consistency of declarations will result in easier code readability. These errors are reported as violations of Item ucs_12.

Reason for rule: Readability.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 12 - If a function has no parameters use ()
  instead of (void)
3:  * /
4: int foo(void);                // UCS12 Violation
```

CodeWizard Output

```
[ucs12.C:5] If a function has no formal parameters, use ()
instead of (void)
```

```
Violation: Universal Coding Standards item 12
```

```
function foo uses (void).
```

Item ucs_13

If, else, while, and do statements shall be followed by a block, even if it is empty

To avoid confusion when reading a piece of code, you should follow loop statements with an empty block rather than with a semi-colon, which is easy to miss. CodeWizard can detect if you do not follow `If`, `else`, `while`, and `do` statements with a block. These errors are reported as violations of Item ucs_13.

Reasons for rule: Readability and maintainability. (See UCS 14.) This rule also helps to prevent errors such as:

```
while(1); { // infinite loop
    do_something();
}
```

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 13 - If, else, while, and do statements shall
3:  * be followed by a block, even if it is empty
4:  * /
5: int main()
6: {
7:     if (1) ;                // UCS13 Violation
8:     return 0;
9: }
```

CodeWizard Output

[ucs13.C:8] If, else, while, and do statements shall be followed by a block, even if it is empty

Violation: Universal Coding Standards item 13

Item ucs_14

If a block is a single statement, enclose it in braces

To avoid causing confusion for people who are reading a piece of your code, you should always enclose a single-statement block in braces. CodeWizard can detect if single-statement blocks are not enclosed in braces. These errors are reported as violations of Item ucs_14.

Reasons for rule: Readability and maintainability. If you are maintaining the code and you see this:

```
if (condition)
    do_something();
```

you may be tempted to update it like this:

```
if (condition)
    do_preparation();
    do_something();
```

instead of like this:

```
if (condition) {
    do_preparation();
    do_something();
}
```

If you write the original code like this:

```
if (condition) {
    do_something();
}
```

there will be less room for error for the maintainer of the code.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 14 - If a block is a single statement,
  enclose it in braces
3:  * /
4: int main()
5: {
6:     if (1)
7:         return 1;           // UCS14 Violation
8: }
```

CodeWizard Output

[ucs14.C:7] If a block is a single statement, it shall be enclosed in braces

Violation: Universal Coding Standards item 14

Item ucs_15

Whenever a global variable or function is used, use the :: operator

CodeWizard can detect if you do not use the :: operator whenever a global variable or function is used. Using the :: operator in these situations will make it easier to determine which variable is being used while code is being read. These errors are reported as violations of Item ucs_15.

Reasons for rule: Readability and maintainability. Using the :: operator helps the maintainer of code to distinguish quickly between global variables and local variables, and it gives the maintainer more freedom with local variable names without colliding with global names.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 15 - Whenever a global variable or function is
   used,
3:  *                               use the :: operator
4:  * /
5: int a = 10;
6: int main()
7: {
8:     int b = a;                               // UCS15 Violation
9: }
```

CodeWizard Output

```
[ucs15.C:9] Whenever a global function or variable is
referenced, use the :: operator
```

Informational: Universal Coding Standards item 15
the following globals are referenced without the ::
operator.

a

Item ucs_16

Do not use public data members

Using public variables is contrary to the principle of data encapsulation and can allow the value of the variable to be changed by any user of the class. CodeWizard can detect if you use public data members. These errors are reported as violations of Item ucs_16.

Reason for rule: Data hiding. If you follow this rule, you will insulate the class interface from the class implementation. You will also provide consistency for users of the class to access data through member functions every time. Internally, the class may change which data types are used to store the data without breaking the interface.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 16 - Do not use public data members
3:  * /
4: class A
5: {
6: public:
7:     int a;                // UCS16 Violation
8: };
```

CodeWizard Output

```
[ucs16.C:6] Do not specify public data members
Violation: Universal Coding Standards item 16
Class A has the following public data members:
a
```

Item ucs_17

If a class has any virtual functions it shall have a virtual destructor

Declaring virtual destructors in classes that have virtual functions causes the compiler to call destructors for each class from which the object inherits. CodeWizard can detect if you write a virtual function without also writing a virtual destructor. These errors are reported as violations of Item ucs_17.

Reason for rule: Prevents memory leaks in derived classes. A class that has any virtual functions is intended to be used as a base class, so it should have a virtual destructor to guarantee that the destructor will be called when the derived object is referenced through a pointer to the base class.

Example

```

1: /*
2:  * UCS item 17 - If a function has any virtual functions
  it shall
3:  *
           have a virtual destructor
4:  * /
5: class A
           // UCS17 Violation
6: {
7: public:
8:     );
9:     virtual int foo();
10: };

```

CodeWizard Output

```
[ucs17.C:7] If a class has virtual functions it shall have a
```

virtual destructor

Severe violation: Universal Coding Standards item 17

class A has virtual functions without a virtual destructor.

Item ucs_18

Public member functions shall return const handles to member data

This item will help you avoid having problems that point to deallocated memory. CodeWizard reports any public member functions that do not return `const` handles to member data as violations of Item ucs_18.

Reason for rule: When you provide `non-const` handles to member data, you undermine encapsulation by allowing callers to modify member data outside of member functions.

Example

```

1: /*
2:  * UCS item 18 - Public member functions shall return
  const handles
3:  *
  to member data
4:  * /
5: class A
6: {
7: public:
8:     int foo();
9:
10: private:
11:     int a;
12: };
13: int A::foo()
14: {
15:     return a;                // UCS18 Violation
16: }
```

CodeWizard Output

```
[ucs18.C:17] Public member functions shall always return  
const handles to member data
```

Severe violation: Universal Coding Standards item 18

Item ucs_19

A class that has pointer members shall have an operator= and a copy constructor

Default memberwise initialization is generally insufficient for classes that contain pointer members because the destructor is invoked for every class object, which may result in dangling references and other program errors. CodeWizard can detect if classes that have pointer members do not also have an `operator=` and a copy constructor. These errors are reported as violations of Item ucs_19.

Reason for rule: Prevents memory leaks and data corruption. If you do not define a copy constructor, the default constructor will be used, which is usually not the desired behavior when a class has pointer members.

Note: This item is suppressed by default.

Example

```
/*
 * UCS item 19 - A class that has pointer members shall have
 * an operator=and a copy constructor
 */

class A
{
public:
    A();
    ~A();

private:
    int *a;
```

```
}; // UCS19 Violation
```

CodeWizard Output

[ucs19.C:7] If a class has pointer members it must define a copy constructor and op=

Violation: Universal Coding Standards item 19

Item ucs_20

If a subclass implements a virtual function, use the virtual keyword

Using the virtual keyword whenever a subclass implements a virtual function will make the code more readable, because the reader will not have to refer back to the base class to see that the function is virtual. CodeWizard can detect if a subclass implements a virtual function without using a virtual keyword. These errors are reported as violations of Item ucs_20.

Reasons for rule: Readability, and more intuitive behavior for deep inheritance hierarchies in that functions that are virtual in a base class will remain virtual in all inherited classes.

Note: This item is suppressed by default.

Example

```

1: /*
2:  * UCS item 20 - If a subclass implements a virtual
3:  * function, use the virtual keyword
4:  * /
5: class B
6: {
7: public:
8:     virtual int foo();
9: };

10: class D
11: {
12: public:
13:     int foo();           // UCS20 Violation

```



```
14: };
```

CodeWizard Output

[ucs20.C:12] If a subclass redefines a virtual function, use the keyword `virtual` in that function's declaration

Violation: Universal Coding Standards item 20

Item ucs_21

Member functions shall not be defined in the class definition

Functions defined within the class definition are implicitly inline. Defining member functions within a class definition will also make the class definition less compact and harder to read. CodeWizard can detect if you define member functions in the class definition. These errors are reported as violations of Item ucs_21.

Reason for rule: Encourages better data hiding by separating interface and implementation.

Note: This item is suppressed by default.

Example

```

1: /*
2:  * UCS item 21 - Member functions shall not be defined in
3:  * the class definition
4:  * /
5: class A
6: {
7: public:
8:     int foo() {};           // UCS21 Violation
9: };10: class D
11: {
12: public:
13:     int foo();             // UCS20 Violation
14: };

```

CodeWizard Output

[ucs21.C:7] Member functions shall not be defined within the class definition

Violation: Universal Coding Standards item 21

class A defines the following functions within the class definition

foo

Item ucs_22

Ellipses shall not be used

Developers should avoid using unspecified function arguments (...) because this deters the strong type checking provided by C++. CodeWizard can detect if you use ellipses. These errors are reported as violations of Item ucs_22.

Reason for rule: Ellipses defeat the benefits of type safety. Use default arguments instead.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 22 - Ellipses shall not be used
3:  * /
4: class A
5: {
6: public:
7:     int foo(int, ...); // UCS22 Violation
8: };
```

CodeWizard Output

```
[ucs22.C:8] Ellipses shall not be used
Violation: Universal Coding Standards item 22
function foo uses ellipses.
```

Item ucs_23

Functions shall explicitly declare their return types

If return types are not explicitly declared, functions will implicitly receive `int` as the return type. However, the compiler will still generate a warning for a missing return type. To avoid confusion, functions that do not return a value should specify `void` as the return type.

CodeWizard can detect if you do not explicitly declare return types in functions. These errors are reported as violations of Item ucs_23.

Reason for rule: Readability. If you do not specify a return type, the compiler will assume either `void` or `int`, depending on the compiler. You should explicitly declare the return type so that the result will be unambiguous.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 23 - Functions shall explicitly declare their
  return types
3:  * /

4: foo();                                // UCS23 Violation
```

CodeWizard Output

```
[ucs23.C:5] Functions shall explicitly declare their return
type
Violation: Universal Coding Standards item 23
```

Item ucs_24

A pointer to a class shall not be converted to a pointer of a second class unless it inherits from the second

CodeWizard can detect if a pointer to a class is converted to a pointer of a second class when the first class does not inherit from the second. Such "invalid" downcasting can result in wild pointers and other disasters. Use virtual function calls instead. These errors are reported as violations of Item ucs_24.

Reason for rule: If you violate this rule, you make assumptions about member layout of classes that may be subject to change, producing memory and data corruption bugs.

Example

```

1: /*
2:  * UCS item 24 - A pointer to a class shall not be
   converted to a pointer
3:  * of a second class unless it inherits from the second
4:  * /
5: class A {};
6: class B {};
7: int main()
8: {
9:     A a;
10:    B b;
11:
12:    b = (b)a;                // UCS24 Violation

```

```
13: }
```

CodeWizard Output

```
[ucs24.C:14] A pointer to a class may not be converted to a  
pointer of a second class unless the first class inherits  
from the second
```

```
Severe violation: Universal Coding Standards item 24
```

Item ucs_25

A pointer to an abstract class shall not be converted to a pointer that inherits from that class

Downcasting from a virtual base is always illegal. CodeWizard can detect pointers to abstract classes that are converted to pointers that inherit from that class. These errors are reported as violations of Item ucs_25.

Reason for rule: When appropriate functions are virtual, converting them is unnecessary and confusing.

Example

```
1: /*
2:  * UCS item 25 - A pointer to an abstract class shall not
   be converted
3:  * to a pointer that inherits from that class
4:  * /
5: class B
6: {
7: public:
8:     int foo() = 0;
9: };
10: class D : public B
11: {
12: public:
13:     int foo();
14: };
15: int main()
16: {
```



```
17:      B b;  
18:      D d;  
19:  
20:      d = (d)b;                // UCS25 Violation  
21: }
```

CodeWizard Output

[ucs25.C:23] A pointer to an abstract class shall not be converted to a pointer of a class that inherits from the abstract class

Severe violation: Universal Coding Standards item 25

Item ucs_26

Do not use the friend mechanism

Using friend functions is usually a sign of an inadequate interface. Fixing the interface, rather than granting friendship, is the best approach. CodeWizard can detect if you use the friend mechanism. These errors are reported as violations of Item ucs_26.

Reason for rule: Using the friend mechanism undermines data-hiding and encapsulation.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 26 - Do not use the friend mechanism
3:  * /

4: int foo();

5: class A
6: {
7:     friend int foo();           // UCS26 Violation
```

CodeWizard Output

```
[ucs26.C:8] Avoid using the friend mechanism
Violation: Universal Coding Standards item 26
Class A has the following friends:
foo
```

Item ucs_27

When working with float or double values, use `<=` and `=>` instead of `==`

Two different paths to the same number won't always lead to the same number. This means, for example, that floating-point numbers that should be equal are not always equal. Using `<=` and `>=` instead of `==` is the recommended solution to the problem.

CodeWizard can detect if you use `==` when working with `float` or `double` values. These errors are reported as violations of Item ucs_27.

Reason for rule: Using `<=` and `=>` helps prevent rounding errors.

Example

```

1: /*
2:  * UCS item 27 - When working with float or double
3:  *    values, use <= and => instead of ==
4:  * /
5: int main()
6: {
7:     float a = 1.0;
8:     float b = 2.0;
9:
10:    if (a == b) {                // UCS27 Violation
11: return 1;
12:    }
13:
14:    return 0;
15: }
```

CodeWizard Output

[ucs27.C:11] When working with float or double expressions,
use less than or equal to or greater than or equal to instead
of ==

Violation: Universal Coding Standards item 27

Item ucs_28

Do not overload functions within a template class

Overloading functions within a template class can lead to problems if the element type appears explicitly in one of them. CodeWizard can detect if you overload functions within a template class. These errors are reported as violations of Item ucs_28.

Reason for rule: Overloading functions may indicate a design flaw in the template class.

Example

```

1: /*
2:  * UCS item 28 - Do not overload functions within a
  template class
3:  * /
4: template <class T> class A
5: {
6: public:
7:     int foo(T);
8:     int foo(int);           // UCS28 Violation
9: };

```

CodeWizard Output

```

[ucs28.C:6] Do not overload functions within a template
class
Severe violation: Universal Coding Standards item 28
The following member functions of class A
foo

```

have potentially conflicting overloaded versions

Item ucs_29

Do not define structs that contain member functions

Member functions should be contained in classes, not structs, because classes support both multiple instances and encapsulation. Structs are also often entirely `public`: whereas the default access level for members and base classes of a class is `private`.

CodeWizard can detect if you define structs that contain member functions. These errors are reported as violations of Item ucs_29.

Reason for rule: Readability. People generally expect types with member functions to be classes, and they still think of structs in terms of their original meaning in C.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 29 - Do not define structs that contain member
  functions
3:  * /
4: struct A
5: {
6: public:
7:     int foo();                // UCS29 Violation
8: };
```

CodeWizard Output

```
[ucs29.C:6] Do not define structs that contain objects with
member functions
```

Violation: Universal Coding Standards item 29
struct A contains the following member functions:
foo

Item ucs_30

Do not directly access global data from a constructor

Directly accessing global data from a constructor is risky because the global object may not yet exist when the "other" static object is initialized. CodeWizard can detect if you directly access global data from a constructor. These errors are reported as violations of Item ucs_30.

Reason for rule: The order of initialization of static objects defined in different compilation units is not defined in the C++ language definition. Therefore, accessing global data from a constructor may result in reading from uninitialized objects.

Example

```

1:  /*
2:   * UCS item 30 - Do not directly access global data from a
   constructor
3:   * /
4:  int a;
5:  class A
6:  {
7:  public:
8:      );
9:  private:
10:      int b;
11:  };
12:  :A()
13:  {
14:      b = a;                // UCS30 Violation
15:  }
```

CodeWizard Output

```
[ucs30.C:8] Do not directly access global data from a  
constructor
```

```
Severe violation: Universal Coding Standards item 30
```

```
Constructor accesses the following global variables:
```

```
a
```

Item ucs_31

Do not use multiple inheritance

Using multiple inheritances can result in strange class hierarchies and less flexible code. Because in C++ there may be an arbitrary number of instances of a given type, it could be that direct inheritance from a class may only be used once (for instance, see the example below). CodeWizard can detect if you use multiple inheritance. These errors are reported as violations of Item ucs_31.

Reasons for rule: Clarity, maintainability. Following this rule prevents ambiguity when classes derive from one object through multiple objects as in the “diamond of death.”

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 31 - Do not use multiple inheritance
3:  * /
4: class B {};
5: class A : public B {};
6: class C : public B {};
7: class D : public A, public C {};// UCS31 Violation
```

CodeWizard Output

```
[ucs31.C:8] Do not use multiple inheritance
Violation: Universal Coding Standards item 31
class D inherits multiply from the following classes:
A
C
```

Item ucs_32

Initialize all variables

A variable must always be initialized before it is used. By always initializing all variables, rather than assigning values to them before they are first used, you can make your code faster and more efficient since no temporary objects are created for the initialization.

CodeWizard can detect if you fail to initialize all variables. These errors are reported as violations of Item ucs_32.

Reason for rule: Following this rule prevents reading from uninitialized variables.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 32 - Initialize all variables
3:  * /
4: int main()
5: {
6: public:
7:     int a;                // UCS32 Violation
8: };
```

CodeWizard Output

```
[ucs32.C:6] Initialize all variables
Violation: Universal Coding Standards item 32
The following variables were not initialized when declared:
a
```

Item ucs_33

All pointers should be initialized to zero

Initializing pointers to zero makes the code more efficient and also helps you catch signals and exceptions. CodeWizard can detect if you do not initialize all pointers to zero. These errors are reported as violations of Item ucs_33.

Reason for rule: Initializing pointers to zero makes it easier to check if a pointer is valid.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 33 - All pointers should be initialized to
  zero
3:  * /
4: int main()
5: {
6:     int *a;                // UCS33 Violation
7: };
```

CodeWizard Output

```
[ucs33.C:6] All pointers should be initialized to zero
Violation: Universal Coding Standards item 33
The following pointers were not initialized when declared:
a
```

Item ucs_34

Always terminate a case statement with break

If the code following a case statement is not terminated with `break`, execution continues after the next case statement, meaning poorly tested code can be erroneous and still appear to work. CodeWizard can detect if you fail to terminate a case statement with `break`. These errors are reported as violations of Item ucs_34.

Reasons for rule: Readability, maintainability.

Note: This item is suppressed by default.

Example

```

1: /*
2:  * UCS item 34 - Always terminate a case statement with
3:  * /
4: int main()
5: {
6:     int a = 0;
7:
8:     switch (a) {
9:         case 0:
10:             return 0;
11:         case 1:
12:             a = 1;           // UCS34 Violation
13:     }
14: }
```

CodeWizard Output

```
[ucs34.C:13] Always terminate a case statement with break  
Violation: Universal Coding Standards item 34
```

Item ucs_35

Always provide a default branch for switch statements

`Switch` statements must always provide a default branch that handles unexpected cases. CodeWizard can detect if you fail to provide a default branch for `switch` statements. These errors are reported as violations of Item ucs_35.

Reason for rule: Maintainability. If all desired cases are handled outside of default, then default can be used for error checking.

Note: This item is suppressed by default.

Example

```

1: /*
2:  * UCS item 35 - Always provide a default branch for
  switch statements
3:  * /
4: int main()
5: {
6:     int a = 0;
7:
8:     switch (a) {
9:         case 0:
10:             return 0;
11:        case 1:
12:            return 1;
13:    }                                     // UCS35 Violation
14: }
```


CodeWizard Output

[ucs35.C:9] Always provide a default branch for switch statements

Violation: Universal Coding Standards item 35

Item ucs_36

Do not use the goto statement

The `goto` statement interrupts the control flow, makes the code more difficult to understand, and has limits on when it can be used. CodeWizard can detect if you use the `goto` statement. These errors are reported as violations of Item ucs_36.

Reasons for rule: Readability, maintainability. Anything written with a `goto` can be written more clearly without a `goto`.

Note: This item is suppressed by default.

Example

```
1: /*
2:  * UCS item 36 - Do not use the goto statement
3:  * /
4: int main()
5: {
6:     goto end;                // UCS36 Violation
7:
8: end:
9:     return 0;
10: }
```

CodeWizard Output

```
[ucs36.C:7] Do not use the goto statement
Violation: Universal Coding Standards item 36
```

Item ucs_37

Provide only one return statement in a function

Minimizing the number of return statements in a function makes the function easier to understand. CodeWizard can detect if you provide more than one return statement in a function. These errors are reported as violations of Item ucs_37.

Reasons for rule: Readability, maintainability, debugging. It is easier to follow the program flow if there is only one return statement.

Note: This item is suppressed by default.

Example

```

1: /*
2:  * UCS item 37 - Provide only one return statement in a
  function
3:  * /
4: int main()
5: {
6:     if (0) {
7:         return 0;
8:     } else {
9:         return 1;           // UCS37 Violation
10:    }
11: }
```

CodeWizard Output

```

[ucs37.C:6] Provide only one return statement in a function
Informational: Universal Coding Standards item 37
```

User Items

(Version 3.x only)

User items are C and C++ coding standards that you have created or that you can customize.

There are two types of user items:

- Coding standards that you have designed in RuleWizard.
- Coding standards written by ParaSoft that can be modified/customized in RuleWizard. These rules are divided into three categories:

- **Naming Convention:** Rules in the 100-199 range are Naming Conventions and have names beginning with "Name." These rules do not represent code which is dangerous or incorrect; rather, they are schemes for naming variables, functions, classes, etc. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

A common element in the Naming Conventions rules is the use of regular expressions. The regular expression format is based on that used in Perl. These regular expressions can be modified to enforce whatever Naming Convention your development group agrees on.

- **Metrics:** Rules in the 600-699 section are Metrics and have names beginning with "Metric." These rules have a slightly different usage than other CodeWizard rules. Whereas most CodeWizard rules are designed to find patterns which represent bad programming practices, the metrics rules are used primarily to produce numbers representing size and complexity of code. Because of this, most outputs in the metrics rules include a "\$count" expression in the Output, which prints the number of occurrences found of that particular pattern. The collectors have count expressions which should be modified according to what cut-off point you are interested in seeing.

For example, `MetricInheritance.rule` reports the number of classes which a given class derives from directly or indirectly. The default is to report this on classes where the ancestor count is greater than 10. If you want a report on the ancestors for every class, set the count expression to `"$$>=0"`. Likewise, if you only want it to report on classes which derive from over 20 base classes, change the count expression to `"$$>20"`.

For a discussion of what metrics are and how to use them, see Mark Shroeder's article, "A Practical Guide to Object-Oriented Metrics" in *IT Pro*, November/December, 1999.

- **C Rules:** Rules in the 700-799 range are rules that were designed especially for C developers.
- **Miscellaneous:** Rules in the 200-599 range are miscellaneous C and C++ coding standards that users can customize.

Violations of user items are labeled "user-<item number>".

You can suppress individual user rules or all user rules as you would suppress rules in any other category.

For more information on creating, modifying, and enforcing custom coding standards, open RuleWizard by typing `rulewizard` at the prompt, then choose **Help> RuleWizard Documentation** in the RuleWizard GUI to open the RuleWizard Users Guide.

For a description of built-in user rules, choose **Help> Rule Documentation** in the RuleWizard GUI.

Rule	Description	Violation	Enabled	C/C++
ArrayElementAccess	Array elements shall be accessed by the array operator []	V	N	C/C++
AssignAllMemberVar	Assign to all member variables in operator= functions	V	N	C++
AssignCharTooHigh	Do not use constants that are greater than the char type's legal range	V	N	C/C++
AssignCharTooLow	Do not use constants that are less than the char type's legal range	V	N	C/C++
AssignUnCharTooHigh	Do not use constants that are greater than the unsigned char type's legal range	V	N	C/C++
AssignUnCharTooLow	Do not use constants that are less than the unsigned char type's legal range	V	N	C/C++
AssignmentOperator	Declare an assignment operator for each class with pointer member variables	V	N	C++

Rule	Description	Violation	Enabled	C/C++
BaseDestructors	Make destructors virtual for all base classes	V	N	C++
BitwiseInCondition	Do not use bitwise operator in conditional	PV	N	C/C++
BreakInForLoop	Avoid breaks in for loops	SV	N	C/C++
CastFuncPtrToPrimPtr	Do not cast a pointer to a function to a pointer of primitive type	V	N	C/C++
CastPointer	Do not cast pointers to non-pointers	V	Y	C/C++
CastUnsigned	Do not cast an unsigned pointer to an unsigned int	V	Y	C/C++
CharCompare	Do not compare chars to constants out of char range	SV	Y	C/C++
CharacterTest	Use the ctype.h facilities for character test	PV	N	C/C++
CommaOperator	The comma operator shall only be used in for statements and variable declarations	V	N	C/C++
ConstParam	Declare reference parameters as const reference whenever possible	V	N	C/C++

Rule	Description	Violation	Enabled	C/C++
ConstPointerFunction Call	Const data type should be used for pointers in function calls if the pointer will not be modified	V	N	C/C++
DeclDimArray	Don't declare the magnitude of an array declaration	PV	N	C/C++
DeclareArrayMagnitude	Don't declare the magnitude of a single dimensional array in the parameter declaration	V	N	C/C++
DeclareBitfield	Do not declare member variables as bit-fields	I	N	C/C++
DeclareExplicit Constructor	Do not use the keyword 'explicit' for a constructor	I	N	C/C++
DeclareMutable	Do not declare member variables with the 'mutable' keyword	I	N	C++
DeclareRegister	Do not declare local variables with the 'register' keyword	I	N	C/C++
DeclareStaticLocal	Do not declare local variables with the 'static' keyword	I	N	C/C++
DeletelfNew	Write operator delete if you write operator new	V	N	C++

Rule	Description	Violation	Enabled	C/C++
DeleteNonPointer	Do not call delete on a non-pointer	SV	Y	C++
DoWhile	Avoid do statements	I	Y	C/C++
EnumKeyword	Do not use an enum keyword to declare a variable in C++	I	N	C++
EnumNamingConvention	In an enumerated list, list members (elements) shall be in uppercase and names or tags for the list shall be in lower-case	PV	N	C/C++
EqualityFloat	Do not check floats for equality; check for greater than or less than	SV	Y	C/C++
ExplicitEnumValues	When using enum, the values of each member should be explicitly declared	PV	N	C/C++
ExplicitLogicalTest	Use explicit logical tests in conditional expressions	V	N	C/C++
ExprInSizeof	Avoid expressions in sizeof operator	V	N	C/C++
FileNameConvention	Use lowercase for file names	PV	N	C/C++
ForLoopVarAssign	Do not assign to loop control variables in the body of a for loop	V	N	C/C++

Rule	Description	Violation	Enabled	C/C++
FractionLoss	Do not assign the dividend of 2 ints to a float	V	Y	C/C++
FuncModifyGlobalVar	Avoid functions that modify the global variable	V	N	C/C++
FunctionSize	Avoid functions over 50 lines	I	N	C/C++
GlobalPrefixExclude	Only use global prefixes for global variables	I	N	C/C++
GlobalVarFound	Avoid global variables	I	N	C/C++
HeaderInitialization	Headers should not contain any initialization	PV	N	C/C++
IfAssign	Avoid assignment in if statement condition	SV	Y	C/C++
IfElse	Give each if statements an else clause	I	N	C/C++
ImplicitUnsignedInit	Do not initialize unsigned integer variables with signed constants	V	N	C/C++
InitCharOutOfRange	Avoid constants out of range for the char type	V	N	C/C++
InitPointerVar	Initialize all pointer variables	V	N	C/C++

Rule	Description	Violation	Enabled	C/C++
InitUnCharOutOfRange	Avoid constants out of range for the unsigned char type	V	N	C/C++
LocalVariableNames	Local variable names shall be proper lowercase	PV	N	C/C++
LongConst	Use capital "L" instead of lowercase "l" to indicate long	SV	Y	C/C++
ManyCases	Avoid switch statements with many cases	I	N	C/C++
MetricBlockofCode	Number of blocks of code per function	V	N	C++
MetricBreakEncap	Number of global variable references per member function	V	N	C++
MetricFuncCall	Number of function calls per function	V	N	C++
MetricInheritance	Number of base classes	V	N	C++
MetricMembers	Number of data members per class	V	N	C++
MetricMethod	Number of methods per class	V	N	C++
MetricParam	Number of parameters per method	V	N	C++
MetricPrivateMembers	Number of private data members per class	V	N	C++

Rule	Description	Violation	Enabled	C/C++
MetricPrivateMethod	Number of private methods per class	V	N	C++
MetricProtectedMembers	Number of protected data members per class	V	N	C++
MetricProtectedMethod	Number of protected methods per class	V	N	C++
MetricPublicMembers	Number of public data members per class	V	N	C++
MetricPublicMethod	Number of public methods per class	V	N	C++
ModifyInCondition	Do not use operator ++ or -- in the conditional part of if, while, or switch	PV	N	C/C++
NameBool	Begin boolean type variable names with 'b'	I	N	C/C++
NameClass	Begin class names with an uppercase letter	I	N	C++
NameConflict	Avoid internal or external name conflict	V	N	C
NameConstantVar	Begin constant variable names with 'c'	I	N	C/C++
NameDataMember	Begin class data member names with 'its'	I	N	C++

Rule	Description	Violation	Enabled	C/C++
NameDouble	Begin double type variable names with 'd'	I	N	C/C++
NameEnumType	Begin enumerated type names with an uppercase letter that is prefixed by the software element and suffixed by '_t_'	I	N	C/C++
NameFloat	Begin float type variable names with 'f'	I	N	C/C++
NameFunction	Begin function names with an uppercase letter	V	N	C++
NameGlobalVar	Begin global variable names with 'the'	I	N	C/C++
NameInt	Begin integer names with 'i'	I	N	C/C++
NameIsFunction	Begin 'is' function names with bool values	I	N	C/C++
NameLongInt	Begin long integer value names with 'li'	I	N	C/C++
NamePointerVar	Prefix variable type pointer names with 'p'	I	N	C/C++
NameShortInt	Begin short integer variable names with 'si'	I	N	C/C++

Rule	Description	Violation	Enabled	C/C++
NameSignedChar	Begin signed character variable names with 'c'	I	N	C/C++
NameString	Begin terminated characters' string variable names with 'sz'	I	N	C/C++
NameStructType	Begin struct type names with an uppercase letter that is prefixed by software element and suffixed by '_t'	I	N	C/C++
NameUnsignedChar	Begin unsigned character type names with 'uc'	I	N	C/C++
NameUnsignedInt	Begin unsigned integer type variables with 'ui'	I	N	C/C++
NameVariable	Begin variable names with a lowercase letter	I	N	C/C++
NamingStructUnionMembers	Use lowercase letters for structure and union member names	PV	N	C/C++
NonScalarTypedefs	Append names of non-scalar typedefs with "_t"	PV	N	C/C++
NumberFunctionParam	Avoid functions with more than 5 parameters	V	N	C/C++

Rule	Description	Violation	Enabled	C/C++
OpEqualThis	Return reference to *this in operator= functions	SV	N	C++
PassByValue	Pass built-in types by value unless you are modifying them	V	N	C/C++
PointerParam Derefence	Don't dereference possibly null pointer parameters	PV	N	C
PublicInterface	Avoid member variables in the public interface	I	N	C++
ReferenceInitialization	Do not initialize a reference to refer to an object whose address can be changed	PV	N	C++
SingularSwitch Statement	Avoid switch statements with only one case	PV	N	C/C++
SourceFileSize	Avoid source files that are longer than 500 lines	I	N	C/C++
SourceNaming Convention	Use the ".c" extension for source file names	I	N	C/C++
StructKeyword	Do not use a struct keyword to declare a variable in C++	I	N	C++
ThrowDestructor	Do not throw from within a destructor	V	Y	C++

Rule	Description	Violation	Enabled	C/C++
TooManyFields	Avoid structs, unions, or classes with more than 20 fields	I	N	C/C++
UnionFieldNotDefined	Define fields for union declarations	I	N	C/C++
UnnecessaryCast	Avoid unnecessary casts	V	N	C++
Unnecessary Equal	Avoid unnecessary “==true”s	I	N	C/C++
UnreachableCode	Do not use unreachable code	PV	N	C/C++
UnusedLocalVariable	Avoid unused local variables	PV	N	C/C++
UnusedParameter	Avoid unused parameters	PSV	N	C/C++
Unused PrivateMember	Avoid unused private member variables	V	N	C++
UsePositiveLogic	Use positive logic rather than negative logic whenever practical	PV	N	C/C++

ArrayElementAccess.rule

Array elements shall be accessed by the array operator []

This rule checks whether array elements are accessed by the array operator [].

Reason for rule: Array elements should be accessed by the array operator [] rather than the dereference operator '*'. Access to array by the dereference operator '*' is cryptic and hides the real intention.

Example

```
void foo()
{
    int array[2];
    array[1] = 0; //OK
    *(array+1) = 0; //Violation
    *array = 0; //Violation
}
```

AssignAllMemberVar.rule

Assign to all member variables in operator= functions

This rule checks whether you assign to all member variables in operator= functions.

Reason for rule: Assigning to all member variables in operator= functions prevents data corruption.

Example

```
class Foo {
public:
    Foo& operator=(const Foo& rhs) {
        if (&rhs == this) {
            return *this;
        }
        x = f.x;
        y = f.y;
        // violation, z is not assigned
        return *this;
    }
private:
    int x;
    int y;
    int z;
};
```

AssignCharTooHigh.rule

Do not use constants that are greater than the char type's legal range

This rule checks whether constants are greater than the char type's legal range.

Reason for rule: The range of legal values for the char type is -128 to 127.

Example

```
/*  
Range of values of char type is -128 to 127.  
*/  
  
void Foo()  
{  
    char c1 = 0;  
    c1 = 145; // hit  
    // ...  
    return;  
}
```

AssignCharTooLow.rule

Do not use constants that are less than the char type's legal range

This rule checks whether constants are less than the char type's legal range.

Reason for rule: The range of legal values for char type is -128 to 127.

Example

```
/*  
Range of values of char type is -128 to 127.  
Violation if value is less than -128.  
*/  
void Foo()  
{  
    char uVal = 0;  
    uVal = -133; // hit  
    // ...  
    return;  
}
```

AssignUnCharTooHigh.rule

Do not use constants that are greater than the unsigned char type's legal range

This rule checks whether constants are greater than the unsigned char type's legal range.

Reason for rule: The range of legal values for char type is 0 to 255.

Example

```
/*
Range of values of unsigned char type is 0 to 255.
Violation if assignment is too high.
*/

void Foo()
{
    unsigned char uVal = 0;
    uVal = 289;
    // ...
    return;
}
```

AssignUnCharTooLow.rule

Do not use constants that are greater than the unsigned char type's legal range

This rule checks whether constants are greater than the unsigned char type's legal range.

Reason for rule: The range of legal values for the char type is 0 to 255.

Example

```
/*
Range of values of unsigned char type is 0 to 255.
Violation if assignment is too low.
*/

void Foo()
{
    unsigned char uVal = 0;
    uVal = -1;
    // ...
    return;
}

/*
```

AssignmentOperator.rule

Declare an assignment operator for classes with pointer member variables

This rule checks whether you declare an assignment operator for classes with pointer member variables.

Reason for rule: Declaring an assignment operator for each class with pointer member variables prevents memory leaks and data corruption.

Note: This rule is similar to item ecpp_11.

Example

```
// See also ECPP 11
class A                                // Violation
{
public:
    A() {}
    ~A() {}
private:
    char *cptr1;
};
```

BaseDestructors.rule

Make destructors virtual for all base classes

This rule checks whether destructors in base classes are virtual.

Reason for rule: Making all destructors for base classes virtual prevents memory leaks and improper object destruction.

Note: This rule is similar to item ecpp_14.

Example

// See ECPP 14

```
class Base
{
public:
    Base() {}
    ~Base() {} // Violation
};

class Derived: public Base
{
public:
    Derived() {}
    ~Derived() {}
};
```


BitwiseInCondition.rule

Do not use the bitwise operator inside conditionals

This rule checks whether bitwise operators are inside conditionals.

Reason for rule: Using bitwise logical operators (&, |, or ~) in the conditional part of `if`, `while` or `switch` can make code difficult-to-read and error-prone.

Example

```
int Foo (int iVar)
{
    if (iVar & 0x01) { // hit
        iVar += 10;
    } else {
        iVar -= 10;
    }
    return iVar;
}
```

BreakInForLoop.rule

Do not use break in for loops

This rule checks if there are `breaks` in your `for` loops.

Reason for rule: Avoiding `breaks` in the `for` loop makes your code easier to follow.

Note: This rule is similar to item `ecpp_14`.

Example

`// Avoid break in for loop to make the code easier to follow.`

```
void func() {  
    for (int i = 0; i < 10; i++) {           // Okay  
    }  
    for (int i = 0; i < 10; i++) {           // Violation  
        if (true) {  
            break;  
        }  
    }  
}
```

CastFuncPtrToPrimPtr.rule

Do not cast a pointer to a function to a pointer of primitive type

This rule checks whether pointers to functions are cast to pointers of primitive types.

Reason for rule: Casting pointers to functions to pointers of primitive types can make code error-prone.

Example

```
void Foo(char *ptrC)
{
    *ptrC = 0;
    return;
}

void f()
{
    void *ptrV = 0;
    void (*funPtr) (char*) = 0;
    funPtr = &Foo;
    ptrV = (void*)funPtr; // violation
    // ...
    return;
}

Output
```

Pointer

CastPointer

Do not cast pointers to non-pointers

This rule checks whether you cast a pointer to a non-pointer.

Reason for rule: Casting a pointer to a non-pointer results in a loss of typechecking.

Example

```
void func()
{
    int i;
    char * pchar = "hello";
    i = (int)pchar;           // Violation
}
```

CastUnsigned.rule

Do not cast an unsigned char to an unsigned int

This rule checks whether you cast an unsigned `char` to an unsigned `int`.

Reason for rule: Casting an unsigned `char` to an unsigned `int` can result in loss of precision.

Example

```
bool foo( char ch )
{
    if( (unsigned) ch == 0xFF ) { // Violation
        return true;
    }
    return false;
}
```

CharacterTest.rule

Use the ctype.h facilities for character test

This rule checks whether you use the ctype.h facilities for character test.

Reason for rule: The ctype.h facilities for character tests and upper-lower conversions (isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper) are portable across different character code sets, are usually very efficient, and promote international flexibility. However, other similarly named functions (such as _tolower, _toupper, _isascii) are not universally portable.

Example

```
#include

void test(char c) {
    if( 'a' <= c && c <= 'z') { //Violation
    }
    if( islower(c) ) { //OK
    }

    while( 'A' <= c && c <= 'Z') { //Violation
    }
    while( isupper(c) ) { //OK
    }
}
```

CharCompare.rule

Do not compare chars to constants out of char range

This rule checks whether you compare `chars` to constants out of `char` range

Reason for rule: Comparing `chars` to constants out of `char` range can result in a constant out of range for the comparison operator.

Example

```
int foobar(char c)
{
    if (c > 300) { return 3;} // Violation
    if (c > 25) { return 2;} // OK
    return 0;
}
```


CommaOperator.rule

The comma operator shall only be used in for statements and variable declarations

This rule checks whether you use the comma operator in `for` statements and variable declarations.

Reason for rule: The comma operator shall only be used in `for` statements, variable declaration(s)/ definitions (s), and multi-statement preprocessor macro definition). Extensive use of the comma operator reduces readability.

Example

```
void foo(int i)
{
    int dVal,count; //OK
    foo((dVal=i,count=dVal+1)); //Violation
}
```

ConstParam.rule

Declare reference parameters as const references whenever possible

This rule checks if you declared your reference parameters as `const` references.

Reason for rule: When your function is not going to modify the argument it is referencing, you should use `const` to protect variables from unintended modifications when the function returns.

Example

```
struct Foo {
    int x;
    int y;
};

int Bar(Foo &f) {           // Violation
    return f.x;
}

int FooBar(Foo &f) {        // Okay
    return f.y++;
}
```

ConstPointerFunctionCall. rule

Const data type should be used for pointers in function calls if the pointer will not be modified

This rule checks whether you use `const` data types for pointers in function calls when the pointer will not be modified.

Reason for rule: The `const` specifier guarantees that the value of the variable cannot be changed. If the value is changed, the compiler will report an error.

Example

```
int foo( int *j) { // Violation
    return  *j;
}

int bar( const int *j) { // Okay
    return  *j;
}
```

DeclDimArray.rule

Don't declare the magnitude of an array declaration

This rule checks if you declare the magnitude of an array declaration.

Reason for rule: When arrays are initialized in the definition, their magnitude should be set by initialization. By allowing the magnitude of an array to be set automatically during definition, changes to the number of elements in the initialization list do not require corresponding changes to the explicit array size.

Example

```
#define SIZE 4

int tab1[SIZE] = {1,2,3}; //Violation
int tab2[]={1,2,3}; //OK
int tab3[SIZE] = {1,2,3}; //Violation
```

DeclareArrayMagnitude.rule

Don't declare the magnitude of a single dimensional array in the parameter declaration

This rule checks if you declare the magnitude of a single dimensional array in the parameters declaration.

Reason for rule: You should not declare the magnitude of a single dimensional array in the argument declaration; if you do, the 'C' language will pass an array argument as a pointer to the first element in the array. In fact, a different invocation of the function may pass array arguments with different magnitudes. Therefore, specifying magnitude of array in a function argument definition might only serve to hinder software maintenance.

Example

```
void fool(int ii[]) { //OK
}

void foo2(int ii[30]) { //Violation
}

void foo3(char a,int ii[30][30][30]) { //Violation
}

void foo4(char a,int ii[][30][30]) { //OK
}
```

DeclareBitfield.rule

Do not declare member variables as bitfields

This rule checks if you declare member variables as bitfields.

Reason for rule: Informational.

Example

```
struct A
{
    int iVarBitField : 8;
    int iVar;
};
```

DeclareExplicitConstructor. rule

Do not use the keyword 'explicit' for a constructor

This rule checks if you use the keyword 'explicit' for a constructor.

Reason for rule: Informational.

Example

```
class X {  
public:  
    explicit X(int);  
    explicit X(double) {}  
};
```

DeclareMutable.rule

Do not declare member variables with the 'mutable' keyword

This rule checks if you declare member variables with the 'mutable' keyword.

Reason for rule: Informational.

Example

```
/*
   This rule is purely informational and will report a mes-
   sage
   when a variable declaration with the 'mutable' keyword is
   found.
*/

class Date{
public:
    int getMonth() const;          // A read-only function
private:
    mutable int month;
    int year;
    int day;
};

int Date::getMonth() const
{
    month++; // Doesn't modify anything if not mutable
    return month;
}
```


DeclareRegister.rule

Do not declare local variables with the 'register' keyword

This rule checks if you declare local variables with the 'register' keyword.

Reason for rule: Informational.

Example

```
void Foo()  
{  
    register int i = 0; // hit  
    return;  
}
```

DeclareStaticLocal.rule

Do not declare local variables with the 'static' keyword

This rule checks if you declare local variables with the 'static' keyword.

Reason for rule: Informational.

Example

```
void Foo()  
{  
    int static iVar = 9; // hit  
    return;  
}
```

DeletelfNew.rule

Write operator delete if you write operator new

This rule checks whether you write operator `delete` when you write operator `new`.

Reason for rule: Writing operator `delete` and operator `new` in concert helps prevent memory corruption and memory leaks.

Note: This rule is similar to item `ecpp10`.

Example

// See ECPP 10

```
class A
{
    // Violation
public:
    A() {}
    void* operator new(size_t size)
    {
        return new int[size];
    }
};
```

DeleteNonPointer.rule

Do not call delete on non-pointers

This rule checks whether you call `delete` on a non-pointer.

Reason for rule: Calling `delete` on a non-pointer results in an invalid operand.

Example

```
class Rhino {
public:
    Rhino();
    Rhino (char *);
    Rhino (const char *);
    operator char *();
};

void func()
{
    Rhino r;
    delete(r); // Violation
}
```

DoWhile.rule

Prefer while statements over do statements

This rule checks whether your code prefers `while` statements over `do` statements.

Reason for rule: Using `do` statements can lead to errors and confusion. Using `while` statements instead of `do` statements can make code clearer and help prevent errors.

Example

```
do {  
    i++;  
} while ( i < 10); // Violation  
  
while (i < 10) { // OK  
    i++;  
}
```

EnumKeyword.rule

Do not use the 'enum' keyword to declare a variable in C++

This rule checks whether your C++ code uses `enum` keywords to declare variables.

Reason for rule: Readability.

Example

```
// enum keyword is unnecessary in C++ when declaring
// a variable.
enum Colors { RED, BLUE, GREEN };
enum Colors c; // Violation, enum keyword is unnecessary
Colors c; // OK
```

EnumNamingConvention. rule

In an enumerated list, list members (elements) shall be in uppercase and names or tags for the list shall be in lowercase

This rule checks naming conventions in enumerated lists; it checks that list members (elements) are in uppercase and that tags for the list are in lowercase.

Reason for rule: The members of a list are equivalent to a defined constant; only the compiler is doing the definition.

Example

```
//OK
enum color {
    RED,
    BLUE,
    GREEN
};

//Violation
enum Color {
    red,
    blue,
    green
};
```

EqualityFloat.rule

Don't check floats for equality; check for greater than or less than

This rule checks whether you check `floats` for equality instead of checking for greater than or less than.

Reason for rule: If you check `floats` for equality, you make your code more susceptible to rounding errors.

Example

```
void func(float a, float b)
{
    if (a==b) { }      // Violation
    while (a!=b) { }   // Violation
    if (a>b) { }       // OK
}
```


ExplicitEnumValues.rule

When using enum, the values of each member should be explicitly declared

This rule checks if the values of each member are explicitly declared when you are using `enum`.

Reason for rule: Declaring values is useful as a documenting feature. Documenting is valuable when using an emulator, logic analyzer or other debugging device.

Example

```
enum my_enum
{
    a,          //Violation
    b = 3       //OK
};
```

ExplicitLogicalTest.rule

Use explicit logical tests in conditional expressions

This rule checks whether you use explicit logical tests in conditional expressions.

Reason for rule: Using explicit logical tests in conditional expressions improves code readability.

Example

```
void foo( int *j)
{
    if(!j){ // Violation
    }
    if(j!=0){ // OK
    }
}
```

ExprInSizeof.rule

Avoid expressions in the sizeof operator

This rule checks for expressions in the `sizeof` operator.

Reason for rule: Expressions in the `sizeof` operator will not be executed.

Example

```
void foo ()
{
    int iVar1 = 0;
    int iVar2 = 0;
    iVar1 = sizeof(iVar2 = 1); /* Violation */

    return;
}
```

FileNameConvention.rule

Use lowercase for file names

This rule checks whether your file names are in lowercase.

Reason for rule: When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

ForLoopVarAssign.rule

Don't assign to loop control variables in the body of a for loop

This rule checks whether you assign to loop control variables in the body of a `for` loop.

Reason for rule: `For` loop control variables should only be modified in the initialization and condition expressions of the `for` loop statement. Modifying them inside the body of the `for` loop makes the loop condition difficult to understand and points to a possible logical flaw in the code.

Example

```
void func() {  
    for (int i = 0; i < 100; i++) { // Violation  
        i += 3;  
    }  
    for (int i = 0; i < 100; i++) { // Violation  
        i++;  
    }  
}
```

FractionLoss.rule

Don't assign the dividend of two ints to a float

This rule checks whether you assign the dividend of two `ints` to a `float`.

Reason for rule: Assigning a dividend of two `ints` to a `float` may cause a loss of fraction.

Example

```
void func()
{
    int a = 3;
    int b = 4;
    double d;
    d = a / b;           // Violation
    d = ((double)a) / b; // OK
}
```

FuncModifyGlobalVar.rule

Avoid functions that modify the global variable

This rule checks whether functions modify the global variable.

Reason for rule: Informational.

Example

```
int theGlob;
// ...
void foo( int iVar )
{
    theGlob = -1;
    if (iVar == 0 )
    {
        theGlob = 0;
    } else {
        theGlob = 1;
    }
    return;
}
```

FunctionSize.rule

Avoid functions with over 50 lines

This rule checks whether functions have over 50 lines.

Reason for rule: Programs should be designed so that most of the functions will be less than 50 lines of source listing. Algorithms are easier to create and to understand if they are built of pieces small enough to be grasped as one concept.

GlobalPrefixExclude.rule

Global prefixes should only be used for global variables

This rule checks whether global prefixes are used with local variables.

Reason for rule: Using global prefixes for local variables reduces code readability.

Example

```
void foo()  
{  
    int theVar = 0;  
    // ...  
    theVar++;  
    return;  
}
```

GlobalVarFound.rule

Avoid global variables

This rule checks whether your code contains global variables.

Reason for rule: Informational.

Example

```
int theVar; // hit
// ...
void foo(int iVar)
{
    theVar = iVar;
    return;
}
```

HeaderInitialization.rule

Headers should not contain any initialization

This rule checks whether headers contain any initialization.

Reason for rule: Headers should not contain any initialization. When initialization is in the header, it is not clear which function "owns" the data (i.e., it doesn't localize the "defining instance"). Multiple source files each including a file containing initializations will generally produce "multiply defined" diagnostics.

IfAssign.rule

Avoid assignment in if statement condition

This rule checks whether your code has assignment within an `if` statement condition.

Reason for rule: Legibility and maintainability; assignment in the context of an `if` statement is easily confused with equality.

Example

```
// Assignment in the context of if statement is easily
// confused with equality.
void foo(int a, int b) {
    if ( a = b ) {} // Violation
    if ( a == b ) {} // OK
}
```

IfElse.rule

All if statements should have an else clause

This rule checks if each of your `if` statements has an `else` clause.

Reason for rule: Writing `if` and `else` in concert improves readability and reliability.

Example

```
void Foo(bool b) {  
    int i = 0;  
    int j = 0;  
    if (b) {           // Informational, should have an else clause  
        i += j;  
    }  
    if (b) {  
        j += i;  
    } else {           // Okay, has an else clause  
        i++;  
    }  
}
```

ImplicitUnsignedInit.rule

Do not initialize unsigned integer variables with signed constants

This rule checks if you initialize unsigned integer variables with signed constants.

Reason for rule: Numeric constants have a type. Use the appropriate suffix to avoid undesirable implicit cast.

Example

```
void foo()
{
    unsigned int x = 21; /* Violation */
    unsigned int y = -21; /* Violation */
    unsigned int z = 21u; /* OK */

    return;
}
```

InitCharOutOfRange.rule

Avoid constants out of range for the char type

This rule checks for constants out of the legal range for the `char` type.

Reason for rule: The legal range of values of `char` type is -128 to 127.

Example

```
void Foo()
{
    char uVal = 157; // hit
    // ...
    return;
}
```

InitPointerVar.rule

Initialize all pointer variables

This rule checks if all of your pointer variables are initialized.

Reason for rule: Initializing pointer variables prevents dereferencing of uninitialized pointers.

Example

```
void foo() {  
    int *i;           // Violation  
    int *k = 0;       // Okay  
}
```


InitCharOutOfRange.rule

Avoid constants out of range for the unsigned char type

This rule checks for constants out of the legal range for the unsigned char type.

Reason for rule: The legal range of values of char type is 0 to 255.

Example

```
void Foo()  
{  
    unsigned char uVal = 259; // hit  
    // ...  
    return;  
}
```

LocalVariableNames.rule

Local variable names shall be proper lowercase

This rule checks whether local variable names are proper lowercase.

Reason for rule: By standardizing the appearance of variables, you can more easily differentiate user-defined variables from constants.

Example

```
void foo() {  
    int Count; //Violation  
    int date;  //OK  
};
```

LongConst.rule

Use capital 'L' instead of lowercase 'l' to indicate long

This rule checks whether you use capital “L” instead of lowercase ‘l’ to indicate `long`.

Reason for rule: Using capital “L” to indicate `long` increases readability and correctness.

Example

```
void func()
{
    int i = 0;
    i = 341; // OK
    i = 34l; // Violation
    i = 34L; // OK
}
```

ManyCases.rule

Avoid switch statements with many cases

This rule checks whether your code contains `switch` statements with many cases.

Reason for rule: Using many `case` statements makes code difficult to follow. More importantly, `switches` with many `cases` often indicate places where polymorphic behavior could better be used to provide different behavior for different types. Note that although the general principle is to avoid many `cases` in a `switch`, the actual cutoff point is arbitrary.

Example

```
void foo(int i) {  
    switch (i) {           // Violation  
        case 1:  
            break;  
        case 2:  
            break;  
        case 3:  
            break;  
        case 4:  
            break;  
        case 5:  
            break;  
        case 6:  
            break;  
        case 7:  
            break;  
        case 8:  
            break;  
    }
```

```
        break;
    case 9:
        break;
    case 10:
        break;
    case 11:
        break;
    default:
        break;
    }
}
```

MetricBlockofCode.rule

Metric: Blocks of code per function

This metric measures the number of blocks of code in a function.

Reason for rule: "Code with many paths will be harder to understand and more likely to contain errors." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, IT Professional, November/December 1999, pg. 34)

Example

```
void func(int i) {  
    if (1) {  
        // do something;  
    } else {  
        // do another;  
    }  
    do {  
    } while(1);  
    switch(i) {  
    case 1:  
        break;  
    default:  
        break;  
    }  
}
```

MetricBreakEncap.rule

Metric: Global variables referenced in member functions

This metric measures the number of global variables referenced in your code's member functions.

Reason for rule: "Global references tend to break encapsulation and inhibit reuse. While global references may be difficult to eliminate entirely, they should be used as sparingly as possible." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, [IT Professional](#), November/December 1999, pg. 33)

Example

```
int globalVar = 1;
class Foo {
public:
    void func() {
        memberVar = GlobalVar;    // Violation, global var is
used here
    }
private:
    int memberVar;
};
```

MetricFuncCall.rule

Metric: Function calls per function

This metric measures the number of function calls in each function.

Reason for rule: "TFC [total function calls] tallies the number of calls to methods and system functions within the system, class, or method. This metric measures size in a way that is more independent of coding style than the LOC [lines of code] metric." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, IT Professional, November/December 1999, pg. 32)

Example

```
class Base {};  
class Dev1 : public Base {};  
class Dev2 : public Dev1 {};
```


MetricInheritance.rule

Metric: Class inheritance level

This metric measures the class inheritance level by calculating the number of base classes.

Reason for rule: “An unnecessarily deep class hierarchy adds to complexity and can represent a poor use of the inheritance mechanism.” (from “A Practical Guide to Object-Oriented Metrics” by Mark Schroeder, IT Professional, November/December 1999, pg. 33)

Example

```
class Base {};  
class Dev1 : public Base {};  
class Dev2 : public Dev1 {};
```

MetricMembers.rule

Metric: Data members per class

This metric measures the number of data members per class.

Reason for rule: "The number of attributes in a class indicates the amount of data the class must maintain in order to carry out its responsibilities." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, IT Professional, November/December 1999, pg. 33)

Examples

```
class Foo {  
    public:  
        int i;  
    protected:  
        int j;  
    private:  
        int k;  
};
```

MetricMethod.rule

Metric: Methods per class

This metric measures the number of methods per class.

Reason for rule: "The number of methods per class indicates the total level of functionality implemented by a class." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, [IT Professional](#), November/December 1999, pg. 33)

Example

```
class Foo {  
    public:  
        void method1();  
    protected:  
        void method2();  
    private:  
        void method3();  
};
```

MetricParam.rule

Metric: Parameters per method

This metric measures the number of parameters in each method.

Reason for rule: "A high number of parameters indicates a complex interface to calling objects, and should be avoided." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, IT Professional, November/December 1999, pg. 34)

Example

```
class Foo {  
public:  
    // Violation so many parameters  
    void func(int i, double d, float f, char c, Foo &);  
};
```

MetricPrivateMembers.rule

Metric: Private members per class

This metric measures the number of private data members per class.

Reason for rule: "The number of attributes in a class indicates the amount of data the class must maintain in order to carry out its responsibilities." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, IT Professional, November/December 1999, pg. 33)

Examples

```
class Foo {  
  private:  
    int k;  
};
```

MetricPrivateMethod.rule

Metric: Private methods per class

This metric measures the number of private methods per class.

Reason for rule: "The number of methods per class indicates the total level of functionality implemented by a class." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, [IT Professional](#), November/December 1999, pg. 33)

Examples

```
class Foo {  
  private:  
    void method();  
};
```

MetricProtectedMembers.rule

Metric: Protected members per class

This metric measures the number of protected data members per class.

Reason for rule: "The number of attributes in a class indicates the amount of data the class must maintain in order to carry out its responsibilities." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, IT Professional, November/December 1999, pg. 33)

Examples

```
class Foo {  
protected:  
    int j;  
};
```

MetricProtectedMethod.rule

Metric: Protected methods per class

This metric measures the number of protected methods per class.

Reason for rule: "The number of methods per class indicates the total level of functionality implemented by a class." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, IT Professional, November/December 1999, pg. 33)

Examples

```
class Foo {  
protected:  
    void method();  
};
```


MetricPublicMembers.rule

Metric: Public members per class

This metric measures the number of public data members per class.

Reason for rule: "The number of attributes in a class indicates the amount of data the class must maintain in order to carry out its responsibilities." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, IT Professional, November/December 1999, pg. 33)

Examples

```
class Foo {  
    public:  
        int i;  
};
```

MetricPublicMethod.rule

Metric: Public methods per class

This metric measures the number of public methods per class.

Reason for rule: "The number of methods per class indicates the total level of functionality implemented by a class." (from "A Practical Guide to Object-Oriented Metrics" by Mark Schroeder, IT Professional, November/December 1999, pg. 33)

Examples

```
class Foo {  
    public:  
        void method();  
};
```

ModifyInCondition.rule

Do not use operator ++ or -- in the conditional part of if, while, or switch

This rule checks whether you use operator ++ or -- in the conditional part of if, while, or switch.

Reason for rule: Using the operator ++ or -- in the conditional part of if, while, or switch can make code difficult-to-read and error-prone .

Example

```
int Foo (int iVar)
{
    if (iVar-- && iVar<10) { // hit
        iVar += 10;
    } else {
        iVar -= 10;
    }
    return iVar;
}
```

NameBool.rule

Begin all boolean type variables with 'b'

This rule checks whether all of your boolean type variables begin with 'b'.

Reason for rule: Hungarian notation. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    bool Var; // Violation  
    bool bVar; // OK  
}
```

NameClass.rule

Begin class names with an uppercase letter

This rule checks whether each class name begins with an uppercase letter.

Reason for rule: When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
class foo {}; // Violation  
class Foo {}; // OK
```

NameConflict.rule

Avoid internal or external name conflict

This rule checks for internal or external name conflict.

Reason for rule: Internal or external name conflicts with a C++ reserved word will cause problems if the program is compiled with a C compiler. Most C compilers do not detect naming conflicts, so name conflicts can lead to unpredictable program behavior.

Example

```
int bool    = 0;
int catch   = 0;
int class   = 0;
int const_cast = 0;
int delete  = 0;
int dynamic_cast = 0;
int explicit = 0;
int export  = 0;
int false   = 0;
int friend  = 0;
int inline  = 0;
int mutable = 0;
int namespace = 0;
int new     = 0;
int operator = 0;
int private = 0;
int public  = 0;
int protected = 0;
int reinterpret_cast = 0;
```

```
int static_cast = 0;  
int template = 0;  
int this = 0;  
int throw = 0;  
int true = 0;  
int try = 0;  
int typeid = 0;  
int typename = 0;  
int using = 0;  
int virtual = 0;  
int wchar_t = 0;
```

NameConstantVar.rule

Begin constant variables with 'c'

This rule checks whether each constant variable begins with 'c'.

Reason for rule: When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
// Constant variable names begin with 'c'  
const int Foo = 0; // Violation  
const int cFoo = 0; // OK
```


NameDataMember.rule

Begin class data member names with 'its'

This rule checks if your class data member names begin with 'its'.

Reason for rule: When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
class Foo {  
  private:  
    int bar;          // Violation  
    int itsBar        // OK  
};
```

NameDouble.rule

Begin all double type variable with 'd'

This rule checks whether all of your `double` type variables begin with 'd'.

Reason for rule: Hungarian notation. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    double Var;    // Violation  
    double dVar;   // OK  
}
```

NameEnumType.rule

Begin enumerated type names with an uppercase letter that is prefixed by the software element and suffixed by '_t'

This rule checks whether the name of each enumerated type begins with an uppercase letter, is prefixed by the software element, and is suffixed by '_t'

Reason for rule: When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
// The names of abstract data types shall begin with an
// uppercase, prefixed by software element and suffixed by
// '_t'.
enum LOC_PossibleColors_t { RED, BLUE, GREEN }; // Violation
enum LOC_PossibleColors { RED, BLUE, GREEN }; // OK
```

NameFloat.rule

Begin all float type variables with 'f'

This rule checks whether all of your `float` type variables begin with 'f'.

Reason for rule: Hungarian notation. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    float Var;    // Violation  
    float fVar;  // OK  
}
```

NameFunction.rule

Begin all function names with an uppercase letter

This rule checks whether each function name begins with an uppercase letter.

Reason for rule: When an entire development group agrees on a common convention, code becomes easier to read and understand.

Example

```
void foo(); // Violation  
void Foo(); // OK
```

NameGlobalVar.rule

Begin global variable names with ‘the’

This rule checks whether each global variable name begins with “the”.

Reason for rule: When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
int global_var; // Violation
int theWindows; // OK
```

NameInt.rule

Begin all integer type variable with 'i'

This rule checks whether all of your integer type variables begin with 'i'.

Reason for rule: Hungarian notation. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    int Var;    // Violation  
    int iVar;  // OK  
}
```

NamelsFunction.rule

Begin 'is' function names with bool values

This rule checks if your 'is' function names begin with `bool` values.

Reason for rule: Beginning 'is' function names with `bool` values improves legibility and clarifies what the return value means.

Example

```
int isPos(int x) {           // Violation
    return x > 0;
}

bool isPositive(int x) {    // Okay
    return x > 0;
}
```


NameLongInt.rule

Begin all long integer variables with 'li'

This rule checks whether all of your `long` integer variables begin with 'li'.

Reason for rule: Hungarian notation. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    long int Var;    // Violation  
    long int liVar;  // OK  
}
```

NamePointerVar.rule

Prefix variable type pointer names with 'p'

This rule checks if your variable type pointer names begin with 'p'.

Reason for rule: When you add a prefix of a type to a variable, the code will be more legible and you will prevent many bugs.

Example

```
int *Foo; // Violation
int *pFoo; // OK
```

NameShortInt.rule

Begin all short integer variables with 'si'

This rule checks whether all of your `short` integer variables begin with 'si'.

Reason for rule: Hungarian notation. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    short int Var;      // Violation  
    short int siVar;    // OK  
    short Violation;    // Violation  
    short siOkay;      // OK  
}
```

NameSignedChar.rule

Begin all signed character variables with 'c'

This rule checks whether all of your signed character variables begin with 'c'.

Reason for rule: Hungarian notation. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    unsigned char Var;    // Violation  
    unsigned char ucVar;  // OK  
}
```

NameString.rule

Begin all terminated characters' string variables with 'sz'

This rule checks whether all of your terminated characters' string variables begin with 'sz'.

Reason for rule: Hungarian notation. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    char Var[] = "\n"; // Violation  
    char szVar[] = "\n"; // OK  
}  
  
}
```

NameStructType.rule

Begin struct type names with an uppercase letter that is prefixed by a software element and suffixed by ‘_t’

This rule checks the prefixes, suffixes, and first letters of your `struct` type names.

Reason for rule: When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
struct LOC_Position { }; // Violation
struct LOC_Position_t {} // OK
```

NameUnsignedChar.rule

Begin all unsigned character type variables with 'uc'

This rule checks whether all of your `unsigned` character type variables begin with 'uc'.

Reason for rule: Hungarian notation. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    unsigned char Var;    // Violation  
    unsigned char ucVar;  // OK  
}
```

NameUnsignedInt.rule

Begin all unsigned integer type variables with 'ui'

This rule checks whether all of your `unsigned` integer type variables begin with 'ui'.

Reason for rule: Hungarian notation. When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    unsigned int Var;    // Violation  
    unsigned int uiVar;  // OK  
}
```


NameVariable.rule

Begin variable names with a lowercase letter

This rule checks if your variable names begin with a lowercase letter.

Reason for rule: When an entire development group agrees on a common convention, code becomes easier to read and more easily understood.

Example

```
void foo() {  
    int Local_var; // Violation  
    int local;    // OK  
}
```

NamingStructUnion Members.rule

Use lowercase letters for structure and union member names

This rule checks whether you use lowercase letters for structure and union member names.

Reason for rule: Using lowercase letters for structure and union member names will make it easier to recognize structure/union members.

Example

```
//OK
struct date_fool {
    int dateMonth;
    int dateDay;
    int dateYear;
};
```

```
//Violation
struct date_foo2 {
    int datemonth;
    int DateDay;
    int Dateyear;
};
```

NonScalarTypedefs

Append names of non-scalar typedefs with "_t".

Reason for rule: Following this convention will make it easier to recognize user-defined data types and maintain the software.

Example

```
class MyClass {  
    int count;  
};  
  
typedef MyClass My_t;    //OK  
typedef MyClass My;      //Violation
```

NumberFunctionParam.rule

Avoid functions with more than 5 parameters

This rule checks for functions with more than 5 parameters.

Reason for rule: Avoiding functions with more than 5 parameters will reduce the amount of coupling between functions. If more parameters are needed, a structure could be used to hold related data and a pointer could be passed.

Example

```
void foo(int a, int b, int c, int d, int e, int f) // Violation
{
}
```

OpEqualThis.rule

Return reference to `*this` in `operator=` functions

This rule checks whether you return reference to `*this` in `operator=` functions.

Reason for rule: Returning reference to `*this` in `operator=` functions protects you from not knowing where the temporary gets destroyed and allows you to declare the `operator=`'s parameter as a reference to `const`, which is safer than just declaring it to be a reference.

Note: This rule is similar to item `ecpp_15`.

Example

// See ECPP 15

```
class A
{
public:
    explicit A(int i = 0) : _i(i) {}
    void operator=(const A& a)           // violation
    {
        if (&a == this) {
            return;
        }
        int _i = a._i;
        return;
    }
private:
    int _i;
};
```

PassByValue.rule

Pass arguments of built-in types by value unless they should be modified by functions

This rule checks if your code passes arguments of built-in types by value, unless they should be modified by functions.

Reason for rule: Passing arguments of built-in types by value improves the efficiency of your code.

Example

```
int Foo(int i, int &j) {           // Violation
    return i + j;
}

int Bar(int i, int &j) {           // Okay
    j += i;
    return j;
}
```


PointerParamDereference. rule

Don't dereference possibly null pointer parameters

This rule checks if you dereference possibly null pointer parameters.

Dereferences of possibly null pointers may be protected by conditional statements or assertions that check that the pointer is not NULL. This rule reports all functions which are passed pointers.

Reason for rule: A dereferenced null pointer is a common cause of program failures.

Example

```
void Foo (int *ptr1, char *ptr2, float *ptrF)
{
    *ptr1 = 10; // possibly null pointer dereference
    ptr2 = 0;
    if( ptrF==0) {
        return;
    }
    *ptrF = 0; // OK
    return;
}
```


PublicInterface.rule

Avoid member variables in the public interface

This rule checks whether your code's public interface contains member variables.

Reason for rule: Legibility and maintainability.

Example

```
class Foo {  
  public:  
    int i;    // Violation  
};  
  
class Foo {  
  private:  
    int i;    // OK  
};
```


ReferenceInitialization.rule

Do not initialize a reference to refer to an object whose address can be changed

This rule checks whether you initialize a reference to refer to an object whose address can be changed.

Reason for rule: The reference to an object in the free store can be deleted via a pointer, and consequently can refer to an object with a null address.

Example

```
void foo()  
{  
    int *ptr = 0;  
  
    int &rptr = *ptr; //Violation  
  
    // assignment through null  
    rptr = 10;  
}
```


SingularSwitchStatement. rule

Avoid switch statements with only one case

This rule checks for `switch` statements with only one `case` statement.

Reason for rule: `switch` statements with only one `case` statement could be described using an `if` statement.

Example

```
void foo ()
{
    int i = 0;
    /*
     ...
    */
    switch(i) /* Violation */
    {
        case 0:  break;
        default: break;
    }

    return;
}
```


SourceFileSize.rule

Avoid source files that are longer than 500 lines

This rule checks for source files that are longer than 500 lines.

Reason for rule: Larger files are more difficult to edit and maintain than smaller files.

SourceNamingConvention. rule

Use the ".c" extension for names of source files

This rule checks whether C source files use the ".c" extension.

Reason for rule: This convention is common practice and is mandated by some compilers.

StructKeyword.rule

Do not use the 'struct' keyword to declare a variable in C++

This rule checks whether your C++ code uses `struct` keywords to declare a variable.

Reason for rule: Using the `struct` keyword to declare a variable improves code readability.

Example

```
// struct keyword is unnecessary in C++ when declaring a
// variable.
struct Position_t {}
struct Position_t Pos1; // Violation
Position_t Pos2;        // OK
```

ThrowDestructor.rule

Do not throw from within destructor

This rule checks whether you throw from within a destructor.

Reason for rule: Throwing from within a destructor may lead to memory leaks and improper object destruction.

Example

```
class Foo
{
public:
    Foo() {}
    ~Foo() {           // Violation
        throw;
    }
};
```


TooManyFields.rule

Avoid structs, unions, or classes with more than 20 fields

This rule checks whether structs, unions, or classes have more than 20 fields.

Reason for rule: Informational.

Example

```
struct _tA
{
    int i0;
    int i1;
    int i2;
    int i3;
    int i4;
    int i5;
    int i6;
    int i7;
    int i8;
    int i9;
    int i10;
    int i11;
    int i12;
    int i13;
    int i14;
    int i15;
    int i16;
    int i17;
    int i18;
```

```
int i19;  
int i20; // more than 20  
int i21;  
int i22;  
int i23;  
};
```

UnionFieldNotDefined.rule

Define fields for union declarations

This rule checks for union declarations that do not have any defined fields.

Reason for rule: Informational.

Example

```
union UNKNOWN // hit
{
};
```

UnnecessaryCast.rule

Avoid unnecessary casts

This rule checks if your code contains unnecessary casts.

Reason for rule: It is not necessary to explicitly upcast to a base class. Removing unnecessary casts makes code easier to read.

Example

```
class Base { /* ... */ };
class Derived : public Base { /* ... */ };
void Func() {
    Derived *d = new Derived();
    Base *b = (Base *)d;           // Violation
    Base *b1 = d;                  // Okay
}
```


UnnecessaryEqual.rule

Avoid unnecessary “==true”s

This rule checks if your code contains unnecessary “==true”s.

Reason for rule: Checking bool expression equality with `true` is unnecessary. Removing unnecessary `== true`s increases legibility and efficiency.

Example

```
bool ret() {  
    return true;  
}  
  
bool ret2() {  
    return ret() == true;    // Violation == is unnecessary.  
}
```


UnreachableCode.rule

Do not use unreachable code

This rule checks for unreachable code in an `if` statement.

Reason for rule: Informational.

Example

```
void foo ()
{
    int i = 0;

    /*
     ...
    */

    if(1) /* Violation */
    {
        i = 1;
    } else {
        i = 2;
    }

    return;
}
```

UnusedLocalVariable.rule

Avoid unused local variables

This rule checks whether any of the declared local variables were not used.

Reason for rule: Eliminating unused local variables increases efficiency and legibility.

Example

```
void func() {  
    int i = 0;    // Possible violation i is declared but not  
    used  
}
```

UnusedParameter.rule

Avoid unused parameters

This rule checks for unused parameters.

Reason for rule: Eliminating unused parameters improves legibility and increases efficiency.

Example

```
int Foo(int i, int k) {           // Violation, k is not used.
    i = 5;
    return i;
}

int Bar(int i, int j) {           // Okay
    i += j;
    return i;
}
```

UnusedPrivateMember.rule

Avoid unused private member variables

This rule checks for unused private member variables.

Reason for rule: Eliminating unused private member variables improves legibility and increases efficiency.

Example

```
class Foo {  
public:  
    Foo(): x(0), y(0) { }  
    int getX() {  
        return x;  
    }  
private:  
    int x;  
    int y;           // Violation, y is declared but not used.  
};
```


UsePositiveLogic.rule

Use positive logic rather than negative logic whenever practical

This rule checks for use of negative logic.

Reason for rule: The use of many logical nots "!" within an expression makes the expression difficult to understand and maintain.

Example

```
void foo( int *j) {  
    if(j!=0){ //Violation  
        (*j)++;  
    }  
    if(j==0){  
    } else { //OK  
        (*j)++;  
    }  
}
```


Index

A

Action field 45

C

Class field 48

CodeWizard

- customizing options 30

- customizing results 43

- installing 4

- results 20

- using 20

- example 15

coding standards

- custom 29

- definition 1

- enforced 65

compilers supported 1

configuration files 30

contacting ParaSoft 12

E

editor, selecting 59

embedded development 1, 23

F

File field 48

I

Insra 53–64

- customizing 61

- troubleshooting 63

installation 4

Item field 46

Items 65

M

makefile 20

N

Notes field 49

P

ParaSoft, contacting 12

Persistence field 46

platforms supported 1

psrc options 30

Q

Quality Consulting 12

R

results 16, 20, 53

- customizing 43

RuleWizard 29

S

source, viewing 59

suppressions 43, 58

- adding 45

- deleting 49

- example 50

- explanation 43

- moving 49

- saving 49

Suppressions Control Panel 45

T

technical support 12
Type field 47

V

violation messages
 deleting 58
 saving/loading 60
 suppressing 43
 viewing 16, 20, 53